UNIVERSITETET I OSLO
Institutt for informatikk

Implementasjon og test av APEX - et rammeverk for adaptiv disk-skedulering med støtte for tjenestekvalitet

Hovedoppgave

Tor Henrik Hanken

1. august 2005

Sammendrag

Multimedia-applikasjoner og andre applikasjoner med høy datagjennomstrømning setter strenge krav til ytelse og krever i mange tilfeller at man kan garantere et bestemt ytelsesnivå for de forskjellige delene av systemet disse applikasjonene kjører på. I et system som kjører slike applikasjoner er det ofte disken som danner flaskehalsen i datastrømmen. Å forbedre ytelsesnivå for disken og å gi mulighet for en fingradert kontroll av hvordan diskressursene brukes blir derfor stadig mer viktig. Disk-skedulereren er mekanismen som holder kontroll på når og hvilke forespørsler som sendes til disken. I denne oppgaven blir disk-skeduleringsrammeverket APEX implementert og testet på Linux versjon 2.6.10, for å vise hvordan disk-skedulering med støtte for forskjellige grader av tjenestekvalitet kan implementeres under realistiske forhold på et ekte operativsystem. Vi viser at går an å lage et system som tar hensyn til fingraderte tjenestekvalitetskrav, og at man kan innføre relativt kompleks håndtering av disk-forespørsler uten nevneverdig tap i ytelsen ellers på systemet.

Forord

Endelig! Veien mot målet har vært lang og tornefull. Jeg står i dyp takknemlighetsgjeld til mine veiledere Pål Halvorsen og Ketil Lund, som har bidratt med inspirasjon og gode innspill underveis i prosessen. Takk til Vera Goebel, som veiledet oppgaven i begynnelsen. Jeg vil også rette en takk til mine kolleger i Copyleft Software AS, som har vært med på å frigjøre tid slik at jeg har kunnet fullføre denne hovedfagsoppgaven. Til slutt vil jeg spesielt takke mine gode venner Asgeir Bjørlykke, Daniel Mikkelsen og Ulrik Einarson for deres fantastiske støtte og oppmuntring under utarbeidelsen av denne oppgaven.

Tor Henrik Hanken, Oslo, 1. august 2005

Innhold

Tabeller

Figurer

Kapittel 1

Introduksjon

1.1 Motivasjon

Dagens operativsystemer (OSer) må håndtere en stor mengde ulike applikasjoner med varierende krav, egenskaper, data-aksessmønster, og så videre. Bare innen klassen av multimedia-applikasjoner er det store variasjoner. Spesielt dersom kontinuerlige multimedia datatyper som for eksempel video, audio og animasjoner er inkludert, kreves det blant annet at store datamengder kan håndteres, gjerne i sanntid. Videre skal dette kanskje kombineres med andre datatyper i en kompleks presentasjon, noe som igjen fører til nye krav.

Multimediadata er normalt svært plasskrevende, og i et slikt system vil derfor lagringssystemet generelt og diskene spesielt, spille en viktig rolle. Lagringssystemet må kunne tilby mange forskjellige tjenesteklasser slik at kravene både fra applikasjonene og fra dataelementer innen applikasjoner kan tilfredsstilles. Siden minne og prossessorkapasitet utvikler seg raskere enn diskkapasiteten blir diskene gjerne den knappeste ressursen i moderne multimediasystemer. Effektiv forvaltning av diskressursene er derfor viktig for å kunne støtte et multimedia-scenario hvor man håndterer presentasjoner bestående av mange forskjellige mediatyper.

I multimediasammenheng er det foreslått flere forskjellige såkalte "mixed-media disk-skedulerere", disk-skedulerere som støtter flere forskjellige klasser av tjenestekvalitet. Mange av disse er imidlertid ikke designet for moderne disker ved at 1) de ikke lenger har tilgang til opplysningene de trenger fra disken, fordi denne mer og mer blir en svart boks hvor det interne skjules for omverdenen, og 2) de er ikke i stand til å utnytte optimaliseringsteknikker som finnes i moderne disker. For å utnytte moderne disker fullt ut bør en disk-skeduleringsalgoritme bare i liten grad basere seg på antagelser om intern informasjon om plasseringen av blokker på disken, og i størst mulig grad overlate den endelige sorteringen av disk-forespørsler til disken selv. Dette kan gjøres ved at man sender flere forespørsler av gangen til disken, slik at den endelige sorteringen kan gjøres av disken.

I tillegg ligger det en utfordring i å få implementert og testet slike systemer i reelle omgivelser. Derfor er da også de fleste mixed-media disk-skedulerere evaluert ved hjelp av simuleringer.

1.2 Problemstilling

På bakgrunn av dette ønsker vi i denne oppgaven å implementere og teste en mixed-media disk-skedulerer i et reelt OS for å vise at ved hjelp av en slik skedulerer vil et system bedre kunne støtte forskjellige applikasjoner og presentere komplekse multimediapresentasjoner. Av eksisterende mixed-media skedulerere er APEX [13] så langt vi vet den eneste disk-skedulerer som er utviklet for bruk sammen med moderne disker, men denne er imidlertid foreløpig kun implementer i et simuleringsmiljø.

I denne oppgaven vil derfor APEX bli implementert og testet i et virkelig OS, Linux versjon 2.6.10. Vi ønsker å vise at APEX er implementerbar (proof-of-concept) i en slik omgivelse og at APEX er i stand til å utnytte optimaliseringsteknikkene i moderne disker. I tillegg ønsker vi å teste APEX med en realistisk arbeidslast.

APEX ble i utgangspunktet designet for å kjøre i user-space, som en del av et Multimedia Database Management System (MMDBMS). Vår implementasjon vil være frittstående, det vil si uavhengig av et DBMS, og vi vil derfor implementere APEX i OS-kjernen. Det vil derfor være nødvendig å gjøre visse modifikasjoner i forhold til det opprinnelige designet: Mens en MMDBMS har kontroll over disklayout, kan implementere sine egne lese- og skrivekall, sitt eget filsystem og kan benytte seg av funksjonalitet på lavt abstraksjonsnivå, vil en applikasjon som benytter seg av en APEX-implementasjon i kjernen ikke ha denne informasjonen. En APEX-implementasjon i kjernen bør kunne benyttes av en hvilken som helst applikasjon, applikasjoner som kun forholder seg til fildeskriptorer som de skriver og leser til. Metadata og kontrollinformasjon om diskforespørsler bør derfor kunne overføres via en uavhengig kanal ned til kjernen som ikke forstyrrer det vanlige lese/skrive-grensesnittet som brukes av user-space-applikasjoner. Applikasjoner som ikke er skrevet for å bruke APEX bør kunne kjøre som normalt.

I tillegg vil vi konsentrere oss om kjernefunksjonene i APEX, det vil si kolli-prinsippet for sende forespørsler til disken og bruk av “token bucket”-prinsippet for allokering av båndbredde.

Siden utarbeidelsen av denne oppgaven ble påbegynt har utviklingen innenfor disk-skedulering på Linux-platformen gått raskt, og vi har i størst mulig grad forsøkt å ta hensyn til denne utviklingen og å bruke disse resultatene under utarbeidelsen av denne oppgaven.

Testene av implementasjonen viser at det er mulig å lage et system som tar hensyn til tjeneste-kvalitetskrav, og at dette kan gjøres uten vesentlig tap i ytelsen for den øvrige trafikken på systemet.

1.3 Oppgavens struktur

- I andre kapittel går vi igjennom multimediasystemer og opplegg for å sikre tjenestekvalitet, samt andre spesielle behov som er spesifikke for multimediasystemer.

- I tredje kapittel behandles disk-skedulering og noen forskjellige algoritmer som er utviklet til dette formålet, vi konsentrerer oss om mixed-media skedulerere.

- Fjerde kapittel omhandler de kravene som stilles til en disk-skedulerer, og hvilke spesielle hensyn som må tas når man innfører støtte for tjenestekvalitet i subsystemet for disk-skedulering i et OS.

- Femte kapittel beskriver hvordan APEX fungerer, og oppsummerer resultatene fra APEX-simuleringene i [13].

- Sjette kapittel beskriver hvordan disk-skedulering i Linux fungerer, og hvordan vi har implementert APEX i Linux 2.6.10.
- I syvende kapittel presenterer vi resultatene fra testene av APEX-implementasjonen, og analyserer resultatene.
- Åttende kapittel er en oppsummering av resultatene og forslag til videre arbeid.

Kapittel 2

Multimedia-applikasjoner og deres systemomgivelser

Multimedia-applikasjoner behandler og spiller av lyd, bilde, video eller andre forskjellige mediatyper samtidig. I dette kapittelet ser vi på hvordan de ulike mekanismene som støtter slike multimedia-applikasjoner er organisert i forhold til hverandre.

2.1 Multimedia-applikasjoner: Et eksempel

For å illustrere hvordan et multimedia-system fungerer kan vi tenke oss et scenario med en sportsportal på Internett som tilbyr video, lyd, kommentarer og løpende resultater fra forskjellige sportsgrener, til et publikum i mange forskjellige land. Brukeren vil her koble seg opp til sportsportalen med en slutt-brukerapplikasjon for å følge sportsbegivenheter ved direkte overføring eller fra opptak. Nettsteder av denne typen er allerede å finne på Internett i dag, for eksempel http://www.sportal.com/. Her er noen av tjenestene en slik multimedia-applikasjon kan tilby:

Avspilling av innhold: Brukeren kan for eksempel velge en spesiell fotballkamp han ønsker å se. Underveis kan han velge å sette avspillingen på pause, eller å spole fremover eller bakover i kampen.

Avspilling av direkte overført innhold: Sportsbegivenheter kan foregå samtidig som brukeren ser på.

Avspilling av sammensatte presentasjoner: Brukeren kan velge om han ønsker dansk eller engelsk kommentatorspor, eller kommentar fra treneren av laget.

Søking i metadata: Den fotballinteresserte brukeren kan for eksempel ønske å finne alle målscoringene til en bestemt spiller, og spille av disse.

Innsjekking av innhold: Lyd, video og andre data skal overføres til sports-tjeneren, og det lagrede materialet må forskynes med riktige metadata som gjør at man kan søke i dem.

Multimedia-applikasjoner som sportsportalen vi skisserte her kjennetegnes ved at de krever litt andre systemomgivelser enn vanlige applikasjoner for at de skal fungere tilfredsstillende for brukeren.

2.2 Multimediadata

En av de tingene som kjennetegner datatypene som brukes til multimediadata er at det dreier seg om store datamengder, i forhold til tradisjonelle datatyper som for eksempel tekst.

En typisk egenskap ved multimedia-datatyper er at de gjerne må prosesseres i henhold til visse tidsfrister, eller at de må prosesseres synkronisert andre media. Eksempelvis kan man ha en film som må spilles av synkront med lyden, og med et visst antall videorammer per sekund, for å unngå hakking i filmen. Et annet eksempel er videospill, hvor man ofte har mange animasjoner som må avspilles samtidig, og bakgrunnsbilder som må utregnes og presenteres på riktig måte i forhold til hverandre innenfor svært korte tidsfrister.

2.3 Elementer i en multimedia-arkitektur

For å spille av en multimediapresentasjon er det en rekke elementer som må være på plass og spille sammen for at alt skal bli riktig. I denne seksjonen presenterer vi de viktigste elementene, med hovedvekt på disken.

2.3.1 Prosessor

Prosessorkraft er viktig i et multimedia-system for å søke i metadata, og for bearbeidelse og sammensetning av presentasjoner. Prosessoren er for eksempel sentral i forbindelse med dekoding og dekryptering av forskjellige multimedia-datatyper som lyd og video. Særlig for animasjoner og komplekse presentasjoner spiller prosessorkraften en viktig rolle.

Prosessorer foretar normalt operasjonene sine relativt raskt i forhold til arbeidet resten av systemet utfører, og det er vanlig å bruke flerprosessorsystemer som deler på arbeidslasten.

2.3.2 Minne

For at en multimedia-presentasjon skal kunne bearbeides og settes sammen på tjeneren må den først lastes inn i minnet. Også i tilfeller med mindre komplekse datatyper går datastrømmen fra disk og gjennom minnet før de sendes videre ut til brukeren.

Siden det tar kortere tid å hente ut data fra minne enn fra disk er det også vanlig å mellomlagre data som aksesseres ofte permanent i minne. Metadata for multimediapresentasjonene kan for eksempel lagres i minnet for å gjøre søking raskere. For datatyper som tar stor plass, som for eksempel video, er det ikke praktisk gjennomførbart å lagre alt i minnet.

2.3.3 Nettverk

Sluttbrukeren og multimediatjeneren er gjerne koblet til hverandre ved hjelp av en nettverksforbindelse. Båndbredde og forsinkelse på denne forbindelsen er avgjørende for hvordan sluttbrukeren opplever systemet. Valget av nettverksprotokoll for overføring av forskjellige typer data til sluttbrukeren vil også avgjøre hvorvidt applikasjonen klarer å levere den riktige kvaliteten på tjenestene.

	Format	Lengde (min)	Oppløsning	Datarate (fps)	Størrelse (MB)	Snitt båndbredde (KB/s)	Maks båndbredde (KB/s)
A	MPEG-2	48	720x576	25	1668.1	582	768
B	MPEG-1	60	320x240	25	348.6	98	192
C	MPEG-2	46	352x288	25	416.1	154	256
D	MPEG-1	29	480x360	25	355.8	212	320
E	MPEG-1	11	720x576	25	221.1	345	228
F	DVD	49	720x576	25	1203.7	421	1088

Tabell 2.1: Eksempler på datamengde og båndbreddebruk i video. (Ref [13])

2.3.4 Disker

Disker er gjerne flere størrelsesordner tregere enn minnet, og spiller derfor en viktig rolle i et system for multimedia-applikasjoner, siden de ofte representerer flaskehalsen i systemet i forhold til andre ressurser som for eksempel prosessorkraft og minne.

Tabell 2.1 gir et bilde på omfanget av dataene og båndbreddebruken ved avspilling av 6 forskjellige videoer.

Vi ser at selv relativt korte videoer tar så stor plass at de ikke kan holdes i minnet i sin helhet, selv på et system som spiller av et begrenset antall videoer. Det blir derfor disken som må sørge for at båndbreddekravene de forskjellige mediene har blir oppfylt.

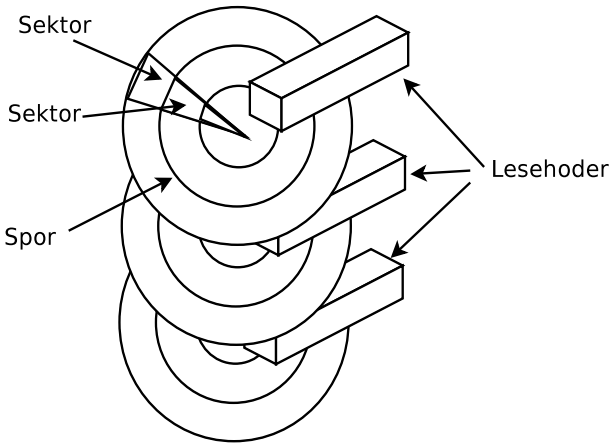
Diskens oppbygning

Figur 2.1 viser hvordan en disk er bygget opp. En disk består av et sett diskplater, hvor hver plate har et lesehode og et skrivehode. Hver plate har et antall spor (tracks), og hvert spor er delt opp i sektorer. En sylinder består av alle sporene som har lik diameter, disse kan leses samtidig siden lese- og skrivehodene på alle platene beveger seg sammen. En diskblokk er grunnenheten for overføring av data mellom minnet og disken, og består av en eller flere etterfølgende sektorer.

Når disken skal lese eller skrive en blokk finner den blokken, plasserer lesehodet, og venter til at disken har gått rundt slik at blokken kommer under lesehodet slik at den kan lese inn blokken og returnere den innleste diskblokken eller statusrapport etter skriving.

Aksesstiden for en disk er tiden det tar fra man ber om en blokk, og til den ligger i minnet. Vi kan regne ut aksesstiden til en disk ved hjelp av følgende formel:

$$st + rf + ot + af$$



Figur 2.1: Diskens oppbygning

Der *st* er søketiden, *rf* er rotasjonsforsinkelsen, *ot* er overføringstiden og *af* er andre forsinkelser.

Søketiden er tiden det tar for å plassere hodet i rett posisjon. Denne tiden kan uttrykkes med følgende formel:

$$\alpha + \beta\sqrt{n}$$

Der α er en tall for en fast overhead, n er antall spor som hodet må bevege seg over, og β er en konstant. Vi bruker kvadratrot fordi hodet aksellererer.

Rotasjonsforsinkelse er tiden det tar før platene har rotert slik at lese/skrivehodet kommer i posisjon over blokkene som skal leses. I gjennomsnitt er dette tiden det tar for platen å foreta en halv rotasjon.

Overføringstiden er tiden det tar for at sektorene under hodet skal roteres så mye at dataene under hodet blir lest. Det som avgjør overføringstiden er tettheten av data per spor, og rotasjonstiden til disken. Overføringstiden kan uttrykkes slik:

$$\frac{ds}{rt}$$

Der *ds* er mengden av data per spor, og *rt* er rotasjonstiden.

Andre forsinkelser er CPU-tid som brukes for å prosessere IO, overføringstid på bussen mellom disk og minne, samt ventetid mens forespørslene står i kø for å bli behandlet. Av disse faktorene er det ventetiden før forespørselen blir behandlet som typisk tar lengst tid.

Diskblokker kan adresseres på to forskjellige måter:

- CHS (Cylinder, Head, Sector) hvor blokken identifiseres ved hvilken sylinder, plate og sektor den befinner seg på.

- LBN (Logical Block Numbers) hvor disken ses som et array av etterfølgende blokker, og en enkeltblokk identifiseres som en indeks i dette diskblokkarrayet.

CHS bruker altså tilsynelatende den fysiske plasseringen av blokken som addresseringsmetode, mens LBN er en logisk adresseringsmetode som ikke antyder noe om blokkens plassering. I praksis er begge addresseringsmetodene logiske: Den nyere utviklingen innenfor diskteknologi gjør at man ikke kan vite noe om blokkens fysiske plassering på grunnlag av hvilken adresse den har på disken.

Moderne diskers uforutsigbarhet

For mange multimedieapplikasjoner vil det være nødvendig å anslå hvor lang tid en diskoperasjon vil ta, slik at ressursene kan fordeles på en hensiktsmessig måte. Moderne disker benytter seg imidlertid av en del teknikker som vanskeliggjør dette anslaget, de skjuler sin indre virkemåte for omverdenen for å kunne foreta optimaliseringer basert på at kontrolleren i disken har full oversikt over det som skjer.

I [13] påpekes disse momentene ved moderne disker som gjør det vanskelig å forutsi hvor lang tid en diskoperasjon vil ta:

Diskblokkaddressering: Selv om CHS tilsynelatende er en fysisk addresseringsmetode er denne logisk, man er aldri garantert at en blokk er fysisk plassert der CHS-adressen dens skulle tilsi. Dermed kan man ikke vite hvor lang tid det vil ta å flytte lesehodet fra en blokk til en annen.

Zone-bit recording: De ytre sporene på en diskplate har større diameter enn de indre, og dermed mer plass til data. Moderne disker benytter seg av dette, og deler opp disken i soner. En sone er et sett sylindre med likt antall sektorer, og siden de har ulik størrelse så vil man oppnå høyere overføringskapasitet hvis en blokk ligger i en av de ytre sonene.

Sparing and slipping: Disker reserverer ekstra sektorer slik at sektorer som blir skadd kan tas ut av bruk og omaddresseres til disse reserveplassene. Diskgeometrien endrer seg altså i løpet av diskens levetid, noe som gjør det enda vanskeligere å vite noe sikkert om den fra utsiden.

Caching og pre-fetching: Moderne disker har et innebygget minne som typisk er på 2-8 MB. Dette minnet ble tradisjonelt brukt for å utjevne forskjellen i hastighet mellom IO-bussen og internt i disken, men nå brukes den også som en cache: Når en blokk leses fra et spor kan disken fortsette å lese dataene som ligger i sporet fordi det er sannsynlig at disse vil bli aksessert i fremtiden (pre-fetching). Tilsvarende kan en skriveforespørsel bli lagret i cachen og utsatt til et senere tidspunkt uten at dette er synlig for omverdenen. Cachen gjør det vanskeligere å anslå hvor lang tid en enkelt diskforespørsel vil ta.

Vi ser altså at moderne disker fungerer som “svarte bokser” som skjuler indre detaljer for omverdenen for kunne gjøre sine egne optimaliseringer. Dette gjør det vanskeligere for utenforliggende enheter å anta noe om hvor lang tid en forespørsel vil ta og om hva som er den optimale arbeidsrekkefølgen for forespørslene.

2.4 Oppsummering

Vi har nå sett på hvilken rolle de forskjellige delene av et multimediasystem spiller, med hovedvekt på disken. Siden disken er såpass sentral, og siden lagringssystemet ofte er en flaskehals, er disk-skedulering helt avgjørende for ytelsen i et slikt system, noe vi skal se nærmere på i neste kapittel.

Kapittel 3

Disk-skedulering

I forrige kapittel så vi blant annet hvilken rolle disken spiller i et typisk multimediasystem. Disken er den ofte den tregeste komponenten i moderne hardware, så det er viktig at man utnytter disse ressursene riktig. Disk-skedulereren sørger for best mulig utnyttelse av båndbredden mellom disken og resten av systemet, og denne båndbredden er en av de knappeste ressursene i systemet.

3.1 Disk-skedulereren

En disk-skedulerer tar imot innkommende forespørsler til disken, og avgjør hvilken rekkefølge disse forespørslene skal sendes videre i. Disk-skedulereren kan ta hensyn til aksessmønster for forespørslene og egenskaper ved de enkelte forespørslene når den foretar en slik utvelgelse, avhengig av hvilket formål skedulereren er laget for å oppfylle. Forskjellige disk-skedulerere tilbyr ulike typer tjenester, og kan ha forskjellige målsetninger. I [13] gis disse hovedegenskapene ved disk-skedulerere:

Allokeringsparadigme: Allokering av båndbredde skjer etter reservasjonsprinsippet eller etter proporsjonal allokering.

I et reservasjonsbasert system vil hver kø av forespørsler få en fast andel av den totale båndbredden som er tilgjengelig.

I et system med proporsjonal allokering har hver kø en vekt, og køens vekt delt på summen av alle køenes vekter er den enkelte vektens andel av båndbredden. Med et slikt opplegg kan man legge til og fjerne køer fordi man automatisk får justert de andre køenes andel forholdsmessig for å gi plass til den nye køen.

Garantinivå: Garantiene en disk-skedulerer gir om når forespørslene behandles kan kategoriseres som deterministisk, statistisk, forbedret best-effort, eller best effort.

Deterministisk garanti vil si at en forespørsel skal behandles i løpet av en fastsatt tid.

Statistisk garanti ligner på deterministisk garanti, men opererer med et statistisk mål på ytelsen for flere forespørsler over tid, for eksempel at gjennomsnittlig behandlingstid for forespørslene skal være et visst antall millisekunder.

Ved allokering basert på forbedret best-effort vil hver kø behandles etter best effort-prinsippet, og hver kø er garantert en viss andel av båndbredden i forhold til de andre køene.

Ved best-effort garanteres ingenting, systemet gjør så godt det kan for å oppfylle forespørslene som kommer inn.

Tjenestetype: En disk-skedulerer kan tilby sanntidstjeneste, tjeneste for høy gjennomstrømning og/eller tjeneste med lav forsinkelse.

Sanntidstjeneste krever at hver enkelt forespørsel behandles innen en viss tid.

Tjeneste for høy gjennomtrømning krever at en viss andel av forespørslene skal behandles med en gitt hastighet.

Lav-forsinkelse-tjeneste er som sanntidstjenester, den krever at hver enkelt forespørsel behandles innen en viss tid. Forskjellen i forhold til sanntidstjenester er at tidsfristen for lav-forsinkelse-tjenester er veldig liten. Dette er typisk interaktiv respons til brukere. For å garantere lav-forsinkelse-tjenester må man reservere en viss andel av båndbredden til slike forespørsler, noe som ville ført til dårlig utnyttelse av ressursene, da slike forespørsler er relativt sjeldne. De fleste disk-skedulerere som støtter lav-forsinkelse tjenester gjør derfor dette ved å utnytte slakk: Man setter av litt mer tid enn det man tror er nødvendig for å behandle et sett forespørsler, og setter inn forespørsler som krever lav forsinkelse direkte i løpet av denne ekstratiden.

Prioritet: Disk-skedulerere kan tilby prioritet på forespørsler, slik at de forespørslene som har høyest prioritet behandles først.

Et viktig poeng i forhold til prioritetssystemer er at det forutsetter at alle applikasjoner som bruker et prioritetshierarki har den samme oppfatningen om prioritet, ellers risikerer man at en applikasjon "overkjører" de andre.

3.2 Klassifisering av disk-skedulerere

Disk-skedulerere kan klassifiseres etter hvilke mål de er designet for å oppfylle. I [16] deles disk-skedulerere inn i ytelsesorienterte, sanntidsorienterte, strømorienterte, og mixed-media-orienterte skedulerere.

3.2.1 Ytelsesorienterte skedulerere

Ytelsesorienterte skedulerere er designet for å øke ytelsen, som oftest ved at de forsøker å se på diskarmens bevegelser for å redusere søketiden. Rene ytelsesorienterte algoritmer som SCAN [7], LOOK [14], C-LOOK og VSCAN [8] organiserer rekkefølgen på håndtering av forespørslene for å maksimere gjennomstrømningen av data. I SCAN beveges lese/skrivehodet fra ytterste sylinder og til den innerste sylinderen og tilbake igjen, og forespørsler tas underveis. C-SCAN ligner på SCAN ved at lese/skrivehodet beveges fra ytre til indre sylinder, men her behandles forespørslene bare når hodet beveger seg i den ene retningen. I LOOK beveges hodet som i SCAN, men snur når det ikke finnes flere forespørsler i den retningen hodet beveger seg. I C-LOOK beveges hodet som i LOOK, men forespørsler tas bare når hodet beveger seg i en av retningene. Disse algoritmene tar bare hensyn til hvor blokkene befinner seg på disken, og tar ikke hensyn til andre egenskaper ved forespørslene som kan gjøre det verdt å gi dem prioritet, alle forespørsler er like viktige.

3.2.2 Sanntidsrienterte skedulerere

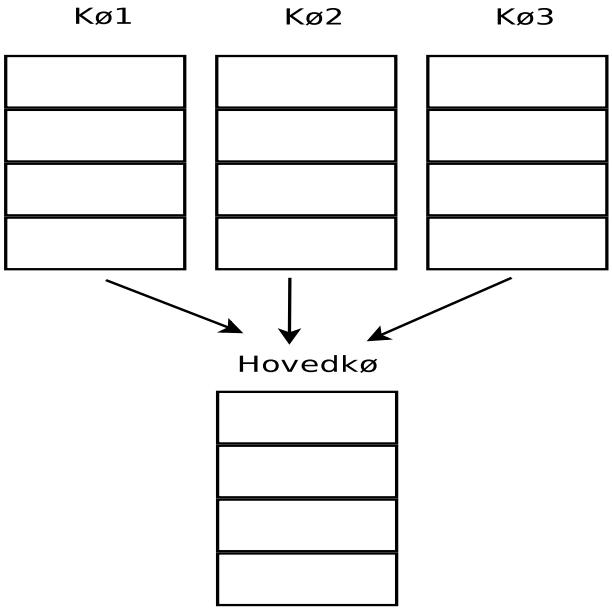
Sanntidsorienterte skedulerere som EDF [11], SCAN-EDF [15] og PSCAN [4] er designet for å støtte deterministiske eller statistiske sanntidstjenester. Disse skedulererne innretter disktrafikken for å unngå at tidsfristene brytes. Forespørsler som ikke har sanntidskrav blir derfor prioritert ned i forhold til de andre forespørslene, og man oppnår gjerne lavere total gjennomstrømning totalt sett i forhold til det man ellers ville oppnådd i med en ren ytelsesorientert skedulerer.

3.2.3 Strømorienterte skedulerere

Strømorienterte skedulerere er laget for å håndtere forespørsler på kontinuerlige datastrømmer som lyd og video, det typiske scenariet er en Video on Demand (VoD) applikasjon. Eksempler på slike algoritmer er CMFS [2], Pre-seeking Sweep algoritm [9], QPMS [18], GSS [20], BubbleUp [5] og T-scan [6]. Strømorienterte skedulerere gir typisk ikke sanntidsgarantier, men opererer med tilgangskontroll og reservasjon som gir statistiske garantier.

3.2.4 Mixed-media skedulerere

Mixed-media skedulerere er designet eksplisitt for å kunne støtte flere tjenestetyper samtidig. For å oppnå dette opererer man gjerne med to lag: Et lag som tar seg av tjenestekvalitet ved å gi prioritet til forespørsler, og et lag som sørger for at de utvalgte forespørslene håndteres mest mulig effektivt. Se figur 3.1 for en illustrasjon av dette prinsippet. Her følger en kort beskrivelse av noen mixed-media skedulerere.



Figur 3.1: 2-lags arkitektur i disk-skedulerere

Fellini

Fellini Storage System [1] støtter to klasser av tjenester, en sanntidsklasse og en klasse for alle andre typer forespørsler. De to klassene er adskilt fra hverandre, og er arbeidsbevarende fordi forespørslene som ikke har sanntidskrav får kjøre på den ledige tiden til sanntidsforespørslene. Fellini støtter altså kun to klasser, og kan ikke omkonfigureres underveis.

Cello

Cello [17] er en mixed-media skedulerer som bruker en kø for hver tjenestekvalitetsklasse, og en endelig kø som som brukes til å mate disken med. En ny forespørsel plasseres i den køen som tilsvarer forespørselens tjenestekvalitet. Deretter har man en klasseuavhengig skedulerer som velger hvor mange forespørsler som skal tas fra hver kø i denne omgangen. Hver kø har så en egen klassespesifikk skedulerer som plasserer sitt tilmålte antall forespørsler i den endelige køen. Til å hjelpe seg i dette arbeidet får de klassespesifikke skedulerne informasjon om den endelige køen fra den klasseuavhengige skedulereren. Denne informasjonen består i at man vet hvor mye slakk hver forespørsel i den endelige køen har på seg, altså hvor lang tid hver forespørsel maksimalt kan vente før den må håndteres. Den klassespesifikke skedulereren har dermed nok informasjon til å sette inn sine forespørsler der de passer best.

Et problem med Cello er at de klassespesifikke skedulerne er opptatt med å sortere forespørslene i sine respektive køer som om disse var de eneste forespørslene som skulle kjøres, det finnes ikke noen mekanisme for å sortere de utvalgte forespørslene globalt. Dette kan medføre et visst effektivitetstap.

Cello er delvis arbeidsbevarende fordi ubrukt båndbredde fordeles blant køene etter deres vekter, men vekten til de forskjellige køene er statisk bestemt, så man kan ikke dynamisk endre på vekten av køene mens systemet kjører. Dette innebærer at den ubrukte båndbredden ikke kan brukes til å jevne ut arbeidsmengden på de forskjellige køene.

Cello baserer seg også på utstrakt kjennskap til diskens geometri siden den regner ut hvor lang tid forespørslene vil ta basert på hvilke diskblokker som er etterspurt og posisjonen til lesehodet. Som vi har sett er trenden for moderne disker at denne typen informasjon skjules fra omverdenen, og algoritmer som baserer seg tungt på denne typen informasjon må ta hensyn til en viss usikkerhet.

MARS

MARS (Massively-parallel And Realtime Storage) [3] støtter flere sanntidskøer og en kø som ikke gir sanntidsgaranti. Til forskjell fra Cello, så har den klasseuavhengige skedulereren her også ansvar for å fordele de utvalgte forespørslene globalt. I MARS kan rundene ha variabel lengde. Arbeidsbesparelse fungerer på samme måte som i Cello, den ekstra båndbredden fordeles på køene etter deres vekt.

Et problem med MARS er at den globale sorteringen ikke åpner for muligheten for at enkelte forespørsler kan hoppe foran i køen, slik at støtten for low-latency tjenester ikke kan sies å være spesielt godt støttet.

Prism

I Prism [19] har de klassespesifikke skedulerne for hver kø ansvar for at de ikke ber om mer båndbredde enn de er tildelt. Hver kø har en statisk fastsatt andel av båndbredden, og lengden på rundene er faste. Prism har også støtte for tilgangskontroll, ved at hver kø sier fra om hva slags båndbredde den vil komme

til å trenge, og en mekanisme som godkjenner eller avslår forespørselen. Prism deler forespørsler i tre grupper: Periodiske, aperiodiske og interaktive forespørsler. For hver runde samles de periodiske og aperiodiske forespørslene og organiseres i grupper. Så settes de interaktive forespørslene inn der det er ekstra båndbredde.

Prism støtter altså deterministisk tjenestekvalitet gjennom de periodiske forespørslene, høy gjennomstrømning gjennom de aperiodiske forespørslene, og best-effort lav-forsinkelse tjeneste.

Båndbredde som ikke blir utnyttet på de interaktive forespørslene går til spille, de fordeles ikke på de andre køene slik som i Cello og MARS.

APEX

APEX [13] er en mixed-media skedulerer som tilbyr dynamisk opprettelse av køer for forskjellige tjenestekvalitetsklasser. Når det kommer inn en ny forespørsel legges denne inn i køen som tilsvarer denne forespørselens tjenestekvalitet, og hvis denne ikke finnes allokeres en helt ny kø for denne klassen.

APEX er en rundebasert skedulerer, og hver runde er like lang. For hver runde plukkes det et antall forespørsler fra hver kø, basert på prinsippet om extended token-bucket: Hver kø får et antall tokens med en gitt rate, og har en grense for hvor mange forespørsler som maksimalt kan velges ut av køen. I hver runde tas forespørsler fra køene basert på dette prinsippet, helt til det maksimale antallet forespørsler per runde er nådd.

I likhet med Cello har APEX en tolagsarkitektur, og prinsippet med å sende flere spørsler samtidig av gangen til diskdriveren er viktig i APEX: Forespørslene som plukkes ut samles opp og sendes videre til diskdriveren i et kolli. Ved å sende flere forespørsler av gangen kan man overlate den endelige sorteringen av diskforespørsler til disken, som har bedre forutsetninger for å avgjøre korrekt rekkefølge i forhold til fysisk plassering.

APEX er arbeidsbevarende: Lengden på hver runde er lik, og denne settes til å være litt lengre enn den tiden det maksimale antallet forespørsler er beregnet til å ta. Denne pausen kan benyttes til å legge inn forespørsler fra en kø som krever lav forsinkelse, for å legge inn forespørsler fra den lengste køen.

3.3 Oppsummering

I dette kapittelet har sett hvordan disk-skedulerere kan klassifiseres, og sett på styrker og svakheter ved forskjellige mixed-media skedulerere utifra hvordan de brukes og hva slags domene de er ment for. I dagens systemer må ofte mange forskjellige applikasjoner med forskjellige krav støttes, og derfor har vi sett nærmere på APEX, siden den støtter flere forskjellige tjenestekvalitetsklasser, og siden den er arbeidsbevarende. I neste kapittel ser vi nærmere på hvilke krav multimedia-applikasjoner stiller til sine omgivelser, med hovedvekt på de kravene som settes til disk-skedulereren, og i kapittel 5 gir vi en mer detaljert gjennomgang av APEX-arkitekturen.

Kapittel 4

Kravanalyse

I de to foregående kapitlene har vi sett på typiske multimedia-applikasjoners systemomgivelser, og skissert virkemåten til forskjellige algoritmer for disk-skedulering.

I dette kapittelet ser vi på hvilke krav multimedia-applikasjoner stiller til sine omgivelser, og på hvordan en implementasjon av disk-skedulering bør fungere for imøtekomme disse kravene.

4.1 Tjenestekvalitetskrav

Som vi så i kapittel 2 har de forskjellige tjenestene multimedia-applikasjoner leverer forskjellige krav i forhold til hva som er nødvendig for at applikasjonen skal fungere tilfredsstillende. Her følger noen eksempler på hvilke forskjellige tjenestekvalitetskrav som er aktuelle i vårt eksempel med en online sports-tjener som tilbyr forskjellige mediatjenester til publikum på forespørsel.

Datarate: Enkelte filformater krever en viss datarate, en MPEG-2 videostrøm med en oppløsning på 720x576 og 25 rammer i sekundet kan for eksempel trenge mellom 582-768 KB/s, se [13]. I sportseksempelet kan man tenke seg at en videooverføring av en viss kvalitet vil kreve at data blir overført fra tjener til sluttbruker med en viss garantert datarate.

Sanntid: Ved sanntids tjenestekvalitet er man garantert at en tjeneste blir levert innenfor et spesifisert tidsrom. I en sports-tjener kan dette for eksempel være direkte overføring av kommentator-lydspor til en fotballkamp, der man må være garantert at lyden kommer frem innenfor en gitt forsinkelses-grense.

Forsinkelse: Forsinkelse er den tiden det tar fra man kommer med en forespørsel og til man begynner å motta svaret, uavhengig av forespørselens størrelse. Lav forsinkelse er viktig i alle situasjoner der brukeren interagerer med systemet. At brukeren trykker på en knapp genererer for eksempel lite datatrafikk, det viktigste er at meldingen kommer raskt frem slik at systemet kan reagere på forespørselen så tidlig som mulig.

Synkronisering: Synkronisering vil si at en presentasjon av ett medie er tidsavhengig av presentasjonen av et annet medie. Et typisk eksempel på dette er synkronisering av tale og bilde når man spiller av en forelesning i et Learning on Demand-system. Et mer komplekst eksempel kan være at tale synkroniseres med en animert presentasjon som består av flere multimediakomponenter.

4.2 Øvrige krav

Som vi så i tabell 2.1 kan selv relativt små medieobjekter ta stor plass, og en multimedie-applikasjon, for eksempel en Video-on-demand server med mange tusen filmer, vil derfor måtte lagre store datamengder. Et OS som skal benyttes til å kjøre multimediasystemer av en viss størrelse bør derfor ha rom for virkelig store lagringssystemer.

Som vi så i tabell 2.1 krever multimedia-applikasjoner som videoavspilling relativt høye datarater, og det er i all hovedsak disken som må sørge for at disse dataratene opprettholdes. Disker er flere størrelsesordner tregere enn minne, og blir derfor gjerne flaskehalsen i datastrømmen. Disken spiller derfor en nøkkelrolle, og i den videre fremstillingen vil vi konsentrere oss om diskens rolle.

4.3 Disk-skeduleringsalgoritmen

Multimedia-applikasjoner består av forskjellige mediatyper med forskjellige krav til tjenestekvalitet. Eksempelvis kan man ha en presentasjon bestående av lyd, video, og tekst, og hver av disse medietypene representerer forskjellige utfordringer i forhold til utnyttelse av disken. En disk-skeduleringsalgoritme som skal utnytte disken maksimalt for å tjene applikasjonen på best mulig måte bør derfor ta hensyn til disse forskjellene. Vi trenger med andre ord en mixed-media disk skedulerer.

Som vi så i kapittel 2 fungerer gjerne moderne disker som "sorte bokser", diskene plasserer data i den rekkefølgen de selv ønsker, og utfører gjerne diskforespørsler i den rekkeføgen de selv finner hensiktsmessig. Programvare som bygger på at man har inngående kjennskap til de interne disposisjonene som gjøres inne i en disk vil gjette feil en del av tiden og dermed ikke dra nytte av alle de optimaliseringene moderne disker utfører.

APEX er en slik skeduleringsalgorime: Den støtter flere tjenestekvalitetsklasser, og serverer kolli som består av flere diskforespørsler til disken på en gang, slik at disken selv kan ta hånd om den interne sorteringen av disse arbeidsoppgavene.

4.4 Disk-skedulererens omgivelser

Det er to typiske måter å disponere over diskressursene som finnes i et multimediasystem. Man kan håndtere disk-skedulering i et program som ligger i user-space, for eksempel et MMDBMS, eller man kan overlate alt til kjernen i operativsystemet.

4.4.1 Disk-skedulering implementert i et MMDBMS

Et MMDBMS som implementerer disk-skedulering i user-space vil ha tilgang til både høynivåinformasjon fra applikasjonene, og lavnivåinformasjon om disken.

Et system som tar hensyn til variasjonene i diskbehov for forskjellige applikasjoner må ha tilgang til lavnivåinformasjon om systemet og må kunne foreta disposisjoner på et nivå som vanligvis er forbeholdt operativsystemet. Å skrive om sluttbrukerapplikasjoner slik at de gjør dette er vanskelig å gjennomføre i praksis. Man kan implementere disk-skedulering i et MMDBMS som ligger imellom sluttbrukerapplikasjonene og OSet og som gir lavnivåinstruksjoner til OSet for å støtte tjenestekvalitet, men dette vil likevel bære preg av at man løfter OS-funksjonalitet ut i user-space. Den mest praksisk interessante

løsningen vil derfor være å implementere en disk-skeduleringsalgoritme i kjernen som støtter forskjellig tjenestekvalitet, og som berører user-space applikasjoner i minst mulig grad.

4.4.2 Disk-skedulering implementert i et OS

En OS-implementasjon vil kun ha tilgang til lavnivåinformasjon, og vil være avhengig av å motta kontrollinformasjon og metadata fra applikasjonen, uten at applikasjonen sitter med denne lavnivå-informasjonen. Man trenger derfor en protokoll mellom applikasjon og disk-skedulerer som tar hensyn til at applikasjonen ikke har detaljinformasjon om disksystemet, og at OSet ikke vet så mye om hvordan applikasjonen opererer. Et problem her er å sørge for at de tjenestekvalitetsparameterne som etterspørres av brukeren blir oversatt til riktige parametre for disk-skedulereren på lavere nivå, slik at disse kravene til tjenestekvalitet blir oppfylt.

4.4.3 Sammenligning mellom OS og MMDBMS-implemententasjoner

Dersom man implementerer disk-skedulering i et MMDBMS har man gode muligheter for å lage et system som fungerer godt, siden applikasjonen har full kontrol over disk-skedulereren. En slik løsning er imidlertid bundet til en enkelt applikasjon.

Dersom man implementerer disk-skedulering på OS-nivå vil enhver applikasjon som kjører på operativ-systemet kunne benytte seg av den, og løsningen blir dermed mer generell. På den annen side vil støtten for tjenestekvalitet være avhengig av at applikasjonen kjenner grensesnittet mot disk-skedulereren, og hvordan parameterne kan oversettes til den enkelte sluttbrukerapplikasjonens høynivåkrav. En OS-implementasjon vil imidlertid kunne eksponere et klart og begrenset grensesnitt som skal kunne brukes av alle typer av applikasjoner, for å begrense behovet for tilpasninger for enkeltapplikasjoner som skal benytte seg av de forskjellige tjenestekvalitetsnivåene.

En implementasjon på OS-nivå vil kreve mindre tilpasninger for enkeltapplikasjoner som skal bruke det. Vi anser at en implementasjon på OS-nivå som eksponerer tilstrekkelig informasjon til applikasjonene vil være å foretrekke.

4.5 Oppsummering

Vi har sett hvilke krav multimedieapplikasjoner stiller til systemene de kjører på:

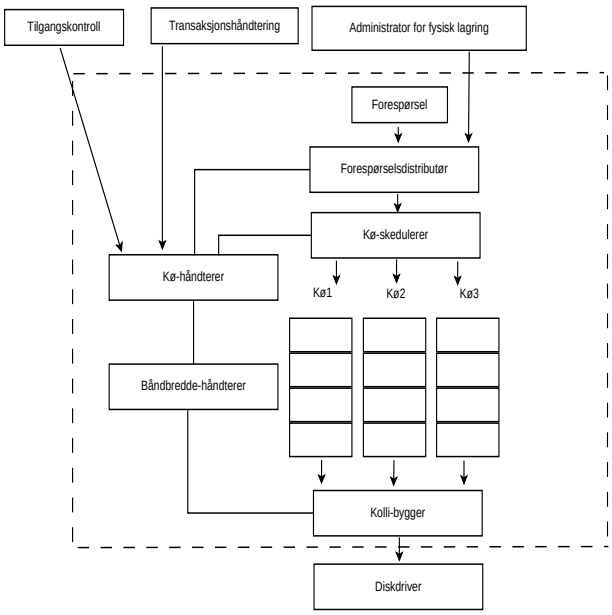
- God utnyttelse av disken er viktig, siden disken gjerne utgjør flaskehalsen i datastrømmen.
- Støtte for alle de forskjellige tjenestekvalitetstypene som multimediaapplikasjoner trenger er ønskelig.
- I et realistisk scenario med store datamengder vil man bruke et operativsystem som støtter slike maskinvarekonfigurasjoner.
- Implementasjon av disk-skedulering på OS-nivå er å foretrekke, fordi det krever færre tilpasninger til applikasjonene man skal benytte.

APEX er en mixed-media disk-skedulerer som er tilpasset moderne disker, og som støtter flere tjenestekvalitetsklasser, og Linux er et operativsystem som støtter store maskinvarekonfigurasjoner. Vi

har derfor valgt å implementere APEX på operativsystemet Linux. I neste kapittel tar vi et mer detaljert gjennomgang av hvordan APEX fungerer.

Kapittel 5

APEX



Figur 5.1: Oversikt over APEX

Dette kapittelet gir en oversikt over disk-skeduleringsrammeverket APEX, som er hovedfokus for denne oppgaven. Motivasjonen for APEX er ønsket om å tilby tjenester med flere forskjellige typer av kvalitet, og et ønske om å lage en skedulerer som utnytter optimaliseringsteknikkene som ligger innebygget i moderne disker. For en mer detaljert teoretisk gjennomgang av arkitekturen i APEX, se kapittel 6 i [13].

5.1 Elementer i APEX

Figur 5.1 viser hvordan de forskjellige komponentene i APEX fungerer sammen. APEX er opprinnelig designet for å bli implementert i user-space i et MMDBMS-scenario der man har komponenter for

tilgangskontroll, transaksjonshåndtering og administrator for fysisk lagring. Disse er tegnet inn utenfor den stripete linjen, siden de ikke er en del av kjernen i APEX. Vi beskriver likevel grensesnittet mot et slikt system.

Forespørselsdistributør: Forespørselsdistributøren mottar diskforespørsler og avgjør hvilken kø forespørselen skal plasseres i.

Kø-skedulereren: Kø-skedulereren mottar en forespørsel og en kø-id fra forespørselsdistributøren, finner riktig kø, og avgjør hvilken plassering forespørselen skal ha innad i køen sin. Plasseringen avgjøres av hva slags kø det dreier seg om: I en sanntidskø vil forespørselen plasseres etter tidsfrist, i en best-effort kø vil forespørselen plasseres bakerst i køen.

Kolli-bygger: Kolli-byggeren plukker forespørsler fra køene og setter sammen et kolli som sendes videre til diskdriveren. Hvor mange forespørsler som plukkes fra hver kø er avhengig av båndbredden denne køen er tildelt. Kolli-byggeren mottar også beskjed tilbake fra diskdriveren hver gang disken er ferdig med en forespørsel.

Kø-administratoren: Kø-administratoren mottar køforespørsler og AdminCommit kall fra tilgangskontrollkomponenten og transaksjonstilstandskall fra transaksjonshåndtereren når APEX er implementert i en MMDBMS-sammenheng. Kø-administratoren sørger for å opprette nye køer, og å fjerne de som ikke er i bruk.

Båndbredde-håndtereren: Båndbredde-håndtereren fordeler den tilgjengelige båndbredden mellom de forskjellige køene, i henhold til båndbredde-reservasjoner og transaksjonstilstander.

Forespørselsdistributøren, køskedulereren og kollibyggeren utgjør forespørselshåndteringen, mens kø-håndtereren og båndbreddehåndtereren utgjør køhåndteringen.

5.1.1 Dynamiske køer

APEX kan opprette og fjerne køer dynamisk under kjøring. Når kø-skedulereren mottar en forespørsel om en ny kø, vil den slå opp i tabellen for forskjellige tjenestetyper og opprette den dersom det godtas globalt av tilgangssystemet. Båndbreddehåndtereren vil så reservere båndbredde for den nye køen. Flere transaksjoner kan dele den samme køen dersom de trenger samme type kø, for eksempel når flere transaksjoner trenger deterministiske sanntidstjenester.

Når en transaksjon er avsluttet og det ikke ligger flere forespørsler i køen vil kø-håndtereren fjerne køen som ikke lenger er i bruk. På denne måten reduseres ekstraarbeidet som er forbundet med dynamiske køer til et minimum.

5.1.2 Utvidet token bucket

APEX går igjennom hver kø i en forutbestemt rekkefølge og plukker forespørsler fra disse som settes sammen til et kolli som skal sendes videre til diskdriveren. For å unngå at de bakerste køene ikke blir utsultet trengs en metode for å avgjøre hvor mange forespørsler som skal plukkes fra hver kø. APEX bruker en utvidet versjon av token bucket-prinsippet til dette formålet.

Prinsippet om Utvidet Token bucket går ut på at hver kø har et antall tokens som kan brukes for å håndtere et antall forespørsler. Et token er en "billett" til å få håndtert en forespørsel. Hver kø vedlikeholder variablene *r* og *b*. *r* bestemmer raten, mens *b* bestemmer dybden på bøtten.

APEX oppdaterer antall tokens i køene basert på raten *r* og tidspunktet for forrige visitt. Om en kø har tildelt en båndbredde på 5 forespørsler i sekundet skal køen tildeles et token hvert 200. millisekund. Dersom en kø har tokens plukker kolli-byggeren ut dette antallet, men aldri flere enn *b* forespørsler, den maksimale dybden på bøtten. Ved å sette *b* unngår man at køene som kommer lenger bak i rekken blir utsultet.

APEX tar også andre hensyn når forespørsler velges ut: Dersom håndtering av en ekstra forespørsel kan føre til at en tidsfrist i en sanntidskø ikke blir overholdt, lar APEX forespørselen bli liggende, selv om køen har tokens.

Støtten for tjenestekvalitet gjennomføres altså ved å tilordne de forskjellige køene passende verdier for *r* og *b*, og ved at forespørsler først plukkes fra sanntidskøene.

5.1.3 Kolli-bygging

For å sette sammen et kolli som inneholder sanntidsforespørsler må man ha et estimat for hvor lang tid det tar å håndtere en enkelt forespørsel. Forespørsler vil nødvendigvis variere i tidsforbruk, så vi vedlikeholder et fast estimat, som kalles *tes*. Størrelsen på *tes* er basert på egenskapene ved disken, blokkstørrelsen og en enkel selvjusteringsmetode som er innebygget i APEX.

Når man setter sammen et kolli er det viktig å vente så lenge som mulig slik at flest mulig disk-forespørsler rekker å nå frem, samtidig som man passer på at disken aldri står uvirksom.

Sammensetning av et kolli foregår på følgende måte:

Finn kontrollerende forespørsel

> For å finne den forespørselen som har den tidligste tidsfristen (*ted*) går vi igjennom det første elementet i alle sanntidskøene. Dersom det ikke finnes noen sanntidsforespørsler brukes rundetidens slutt som *ted*.

Størrelsen på kolli

> For å regne ut størrelsen på kolli:
>
> $$B = \frac{ted - tstart}{tes}$$
>
> hvor
>
> $$tstart = tbas + tes$$
>
> og *tbas* er tiden da sammensetningen startet. Når vi regner ut *B* antar vi alltid det dårligst tenkelige utfallet, at den kontrollerende forespørselen håndteres sist.

Sammensetning av kolli

> Kolliet settes sammen ved at vi går igjennom hver kø, oppdaterer antall tokens for køen, og reduserer *B* alt ettersom hvor mange forespørsler som blir plukket ut.

Når man skal stoppe

Sammensetningen av et kolli avsluttes når *B* er 0, eller når alle køene har blitt besøkt. Deretter regnes fullføringstidspunktet for kolliet, tes, ut, og kolliet sendes til diskdriveren.

5.1.4 Arbeidsbevaring

Med en variabel arbeidsmengde vil det av og til oppstå situasjoner der de køene som har reservert en viss båndbredde ikke utnytter denne fullt ut, mens køer som ikke har reservert båndbredde eller som har lavere båndbredde står igjen med forespørsler i køen selv om alle køene har blitt besøkt. APEX tar hensyn til dette ved å gå inn i en arbeidsbevarende fase dersom *B* er større enn null etter at alle køene har blitt besøkt.

Med token bucket-systemet er det bare de køene som har reservert en viss båndbredde som mottar tokens, slik at beste-forsøk-køer som ikke har reservert båndbredde må bruke den arbeidsbevarende fasen til å få sine forespørsler igjennom.

[13] foreslår flere alternative måter dette kan gjøres på:

- Generell

  Forespørsler plukkes fra alle køene i samme rekkefølge som før, men nå uten at det tas hensyn til antall tokens.

- Behovsbasert

  Køene besøkes igjen, men nå etter et bestemt sorteringskriterie, for eksempel at de lengste køene besøkes først.

- Dedikert

  Den ubrukte båndbredden fordeles på en eller flere utvalgte køer, og hvis disse køene blir tømt fordeles resten av båndbredden etter et av prinsippene over, generelt eller behovsbasert.

5.1.5 Tjeneste med lav forsinkelse

Vi vedlikeholder et estimat, *tef*, som sier hvor lang tid det vil ta å håndtere et kolli. Dette estimatet er konservativt, så vi vil ha et visst slakk, som vi regner ut ved å se på

*tef - (tes * antall gjenværende forespørsler)*

Dersom vi har slakk setter vi inn en forespørsel fra en spesiell kø for lav forsinkelse-tjenester direkte. Forespørsler fra lav forsinkelse-køen settes i gjennomsnitt inn når kolliet er halvveis ferdig, så gjennomsnittlig ferdigstillingstid for en slik forespørsel blir

$$\frac{B}{4 * tes}.$$

5.2 Faktorer som påvirker ytelsen

Ytelsen i APEX påvirkes av en del faktorer. Her ser vi hvordan plassering av data, størrelsen på blokkene, rundetiden og egenskaper ved disken virker inn.

5.2.1 Plassering av data på disken

Et hovedpoeng i som APEX benytter seg av er at forespørslene i et kolli ikke må sorteres før de sendes til disken, siden disken kan bedre forutsetninger for å gjøre denne sorteringen selv.

Likevel er *tes*, det estimerte tiden en forespørsel vil ta, avhengig av plassering av diskblokkene som etterspørres. Det tar typisk kortere tid å hente flere diskblokker som ligger etter hverandre, slik tilfellet gjerne er for video og lyd, enn dersom blokkene ligger spredt utover disken, slik tilfellet kan være for mindre filer som tekst.

tes bør derfor justeres etter hva slags datatyper som ligger lagret og som etterspørres ved normal arbeidsbelastning.

5.2.2 Diskblokkstørrelse

Diskblokker kan være mellom 2K og 64K i det vanlige tilfellet. Mindre blokker betyr at man kan skrive eller hente et større antall blokker, men innebærer også at disken kan overføre mindre data, slik at diskutnyttelsen blir lavere. Med større blokker kan man overføre større mengder data, på bekostning av større forsinkelse per blokk. Når man velger blokkstørrelse er dette en avveining man gjør på bakgrunn av hva slags data som ligger på disken. Et system med store filer kan dra nytte av større diskblokker, mens man gjerne vil velge en mindre diskblokkstørrelse i et system med mange og små filer.

For APEX betyr diskblokkstørrelsen at *tes* bør reduseres litt ved mindre blokkstørrelser i forhold til et scenario med store blokker. Den gjennomsnittlige søketiden for å hente data vil imidlertid være den samme, forskjellen vil ikke være vesentlig.

5.2.3 Rundetid

Rundetiden har direkte innvirkning på hvor mange forespørsler som kan sendes til disken i hver runde. Lengre rundetider betyr bedre utnyttelse av disken, men større forsinkelse ved behandling av forespørsler.

Fordelene ved å sende flere forespørsler til disken av gangen er avhengig av at kolliet har en viss størrelse, og med svært korte rundetider vil denne effekten reduseres. APEX er med andre ord avhengig av at rundene ikke er altfor korte.

5.2.4 Egenskaper ved disken

Ytelsen på disker er avhengig av egenskaper som søketid, rotasjonstid og overføringstid. I tillegg kan disker mellomlagre data som er hentet ut før, eller forhåndslese data som ligger nær de blokkene som nettopp har blitt lest.

I et kolli med forespørsler vil overføringstiden derfor variere svært fra forespørsel til forespørsel. Den gjennomsnittlige overføringstiden for et kolli av en viss størrelse vil likevel holde seg relativt konstant, så disse faktorene vil ha liten innvirkning på ytelsen i APEX.

5.3 Grensesnitt

Som sagt er APEX designet med tanke på å inngå som en del av et større MMDBMS-system, og grensesnittet er laget med dette for øyet. Alle forespørslene inngår derfor i transaksjoner, og en transaksjon kan

være sammensatt av flere sett med forespørsler som kan tilhøre forskjellige køer: Når man skal lese ut en video og underteksten til denne skal de eksempelvis leses samtidig, så de inngår i samme transaksjon, men tjenestekvalitetskravene for de to mediene er forskjellig.

Grensesnittet mellom APEX og omgivelsene består av fire deler:

RequestQ(TrxID, queueName, bandwidth, weight)

AdmitCommit(TrxID, 'OK')

*Schedule(*buf, QID, deadline, priority, TrxID)*

TrxState(TrxID, state, newValue)

For å opprette en kø kalles *RequestQ* med transaksjonsid, typen på køen og enten båndbredde-reservasjon eller vekt, avhengig av hva slags kø som skal opprettes. Resultatet av dette kallet er at det opprettes et transaksjonsobjekt som lagres i transaksjonstabellen. Samtidig foretas en tentativ reservasjon av den tilmålte båndbredden.

Når komponenten for adgangskontroll kaller *AdmitCommit* med transaksjonsid og resultat, informeres APEX om at denne transaksjonen skal gjennomføres, eller at den skal forkastes. Dersom den skal gjennomføres, vil alle de tentative reservasjonene fra *RequestQ* bli satt ut i livet.

For å legge inn en ny forespørsel kalles *Schedule*. Parameterne angir kø-id og transaksjonsid, samt tidsfrist eller prioritet for forespørselen, avhengig av hva slags kø det dreier seg om. Returverdien er *OK* eller *FAIL* avhengig av om forespørselen kunne settes inn i køen.

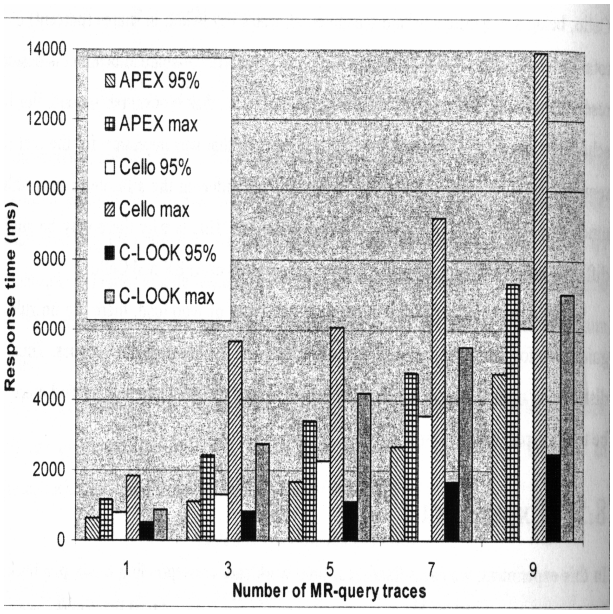
TrxState kan brukes for å starte eller stoppe en transaksjon, eller for å endre reservert båndbredde eller vekt som skal allokeres til transaksjonen.

5.4 Resultater og erfaringer fra APEX

For å vise litt av hva vi forventer av en APEX-implementasjon, har vi tatt med noen av testresultatene i [13]. Her ble APEX implementert og testet i simuleringsmiljøet DiskSim, og sammenlignet med simulerings-implmentasjoner av to forskjellige disk-skedulerere: Den ytelsesorienterte skedulereren C-LOOK, og Cello, som støtter forskjellige klasser av tjenestekvalitet. Gjennom fire eksperimenter med forskjellig arbeidslast ble APEX testet og sammenlignet med de andre skedulererne.

I eksperiment 1 ble avspilling av 6 forskjellige videoer simulert. De seks videoene spilles av med sanntidskrav i Cello og APEX. Ved siden av videoavspillingen ble det sendt over en arbeidslast bestående av metadataspørringer. Trafikken bestående av metadataspørringer ble så variert for å se hvordan respons-tiden for de forskjellige skedulererne ble påvirket, og for å se om den økte arbeidslasten førte til at tids-frister ble brutt. Figur 5.2 viser fordelingen av responstider for forskjellig grad av metadata-trafikk. Som vi ser gjør C-LOOK det best for 95% av forespørslene, med APEX på andreplass og Cello sist. APEX gir altså ikke de beste responstidene på flesteparten av forespørslene. I forbindelse med dette eksperimentet ble det også undersøkt hvorvidt de forskjellige skedulererne brøt sine tidsfrister. Tabell 5.1 viser at APEX og Cello holder sine frister for sanntidsforespørslene, mens C-LOOK bryter disse ganske ofte, til tross for at C-LOOK gir lavere responstid på flesteparten av forespørslene sett under ett.

I eksperiment 2 beholdes samme arbeidslast, men i tillegg legges det inn en utsjekkingsoperasjon, som får 25% av båndbredden under Cello og APEX. Også i dette eksperimentet har APEX lavere



Figur 5.2: Eksperiment 1 - Fordeling av responstider APEX, Cello og C-LOOK (ref [13])

Skedulerer	Antall brudd	Gjennomsnitt	95%	Maksimum
APEX	0	-	-	-
Cello	0	-	-	-
C-LOOK	3266	769ms	2186ms	4626ms

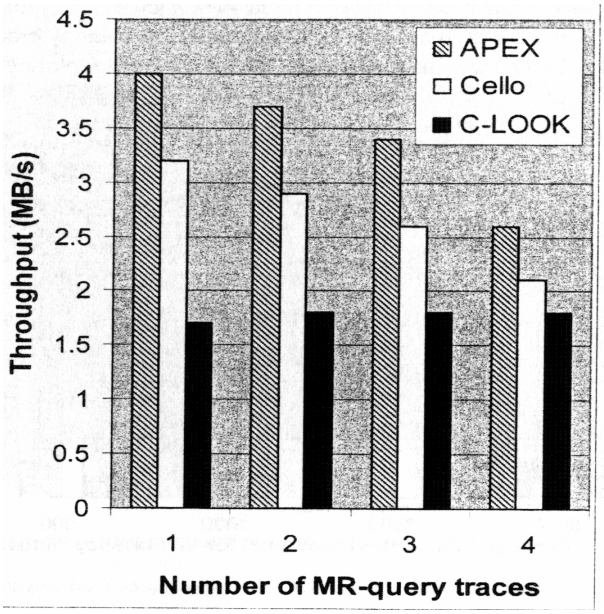
Tabell 5.1: Eksperiment 1 - Tidsfrist-brudd for sanntidsforespørsler (ref [13])

responstider enn Cello, mens C-LOOK har de laveste responstidene. Figur 5.3 viser gjennomstrømningen for utsjekkingsoperasjonen i MB/sekund. Her ser vi at APEX gir høyest gjennomstrømning av de tre skedulererne, med Cello på annenplass, og med C-LOOK sist. Nok en gang ser vi at C-LOOK bryter tidsfristene for sanntidsforespørslene, mens APEX og Cello holder disse, se tabell 5.2.

I eksperiment 3 undersøkte man de forskjellige skedulerernes evne til å håndtere forespørsler med lav forsinkelse. Figur 5.4 viser resultatene fra dette eksperimentet, der APEX oppnådde lavest responstid av alle, med C-LOOK på andreplass og Cello til sist.

I eksperiment 4 kombineres alle arbeidslast-typene: Lav-forsinkelse, utsjekking, metadata-spørringer og sanntidsforespørsler i to forskjellige konfigurasjoner. I den første konfigurasjonen benyttes en sanntidsklient, i den andre konfigurasjonen benyttes fire sanntidsklienter. Figur 5.5 viser responstidene for lav-forsinkelse og metadataspørringer, som vi ser gir APEX lav responstid både ved forespørsler med lav-forsinkelse, og ved metadata-spørringer. Verken Cello eller APEX brøt tidsfristene til sanntidsforespørslene, mens C-LOOK kun brøt tre. Tabell 5.3 viser gjennomstrømning ved utsjekkingsoperasjoner i første og andre konfigurasjon. Vi ser at APEX gjør det godt i begge de to konfigurasjonene etterfulgt av Cello, og med C-LOOK til sist.

Figur 5.6 viser fordelingen av responstider under andre konfigurasjon, med fire sanntidsklienter. Som



Figur 5.3: Eksperiment 2 - Gjennomstrømning for utsjekkingsoperasjon (ref [13])

Skedulerer	Antall brudd	Gjennomsnitt	95%	Maksimum
APEX	0	-	-	-
Cello	0	-	-	-
C-LOOK	1246	540ms	1422ms	2779ms

Tabell 5.2: Eksperiment 2 - Tidsfrist-brudd for sanntidsforespørsler (ref [13])

vi ser gjør APEX det bedre enn Cello. C-LOOK gjør det ganske bra, men har større spennvidde mellom 95% av forespørslene og maksimalverdiene. Også her unngikk APEX og Cello å bryte tidsfrister, mens C-LOOK brøt 53.

I [13] oppsummeres disse resultatene slik:

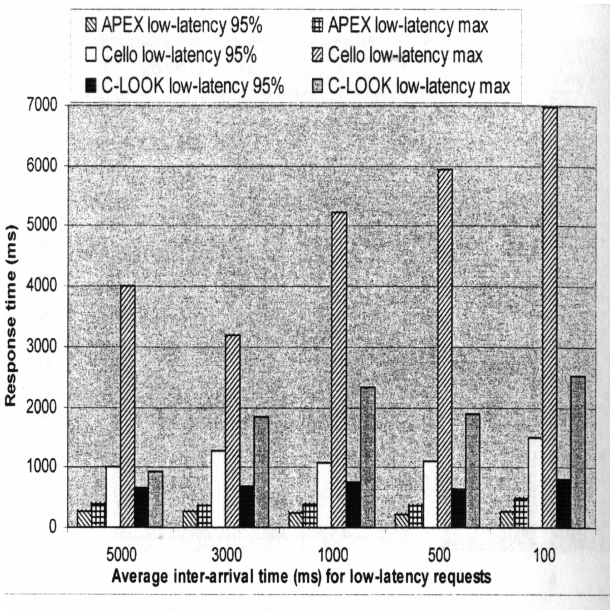
- APEX egner seg til bruk i en MMDBMS

  Scenariene i eksperiment 4 er representative for metadataspørringene som man typisk har i en MMDBMS, og viser at APEX håndterer denne typen arbeidslast bedre enn APEX og Cello.

- APEX egner seg godt i mange forskjellige sammenhenger

	APEX	Cello	C-LOOK
1 sanntidsklient	5.7	4.4	2.7
4 sanntidsklienter	4.6	3.6	2.5

Tabell 5.3: Eksperiment 4: Gjennomstrømning for utsjekkingsoperasjoner i MB/s (ref [13])



Figur 5.4: Eksperiment 3 - Fordeling av responstider APEX, Cello og C-LOOK (ref [13])

De fire eksperimentene viser at APEX kan håndterer mange forskjellige typer arbeidslast på en god måte.

- APEX støtter både høy gjennomstrømning og tjenestekvalitet

  Simuleringene viser at APEX gir bedre gjennomstrømning og kortere responstider enn Cello når det gjelder best-effort tjenestene. Sammenlignet med C-LOOK gir APEX nesten like god utnyttelse av båndbredden, samtidig som APEX overholder de samme sanntidsgarantiene som Cello. Med andre ord er kostnaden lav for å innføre tjenestekvalitet med APEX.

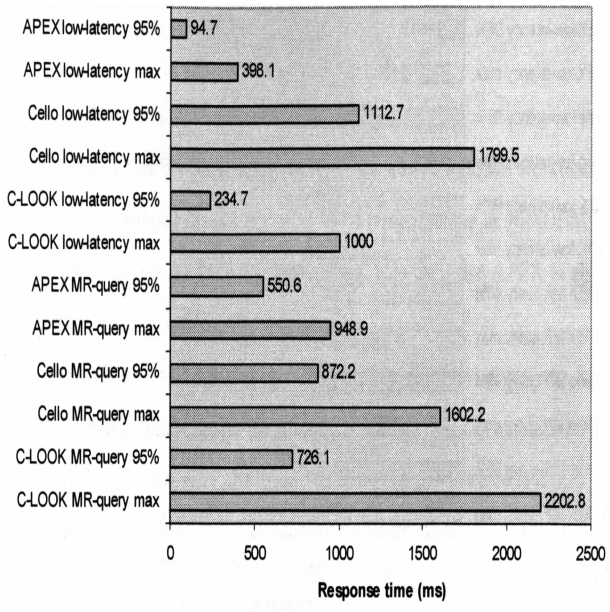
- Disk-skedulering er nødvendig for å støtte tjenestekvalitet

  Simuleringene viste at selv om C-LOOK ga høy ytelse, er de store variasjonene i responstid vanskelig å forene med et system som skal forskyne brukerne med garantier om tjenestekvalitet.

5.5 Oppsummering

I dette kapittelet har vi sett at APEX består av følgende grunnprinsipper:

- Flere forespørsler sendes samtidig til disken, siden dette gir moderne disker et bedre utgangspunkt for å optimalisere operasjonene sine.
- To-lags arkitekturen hvor hver tjenestekvalitetsklasse er representert med hver sin kø, og en mekanisme for å velge ut hvilke forespørsler som skal være med i neste kolli.
- Utvidet token bucket, som gjør at man kan styre utvelgelsesprosessen med enkle kvantifiserbare parametre.

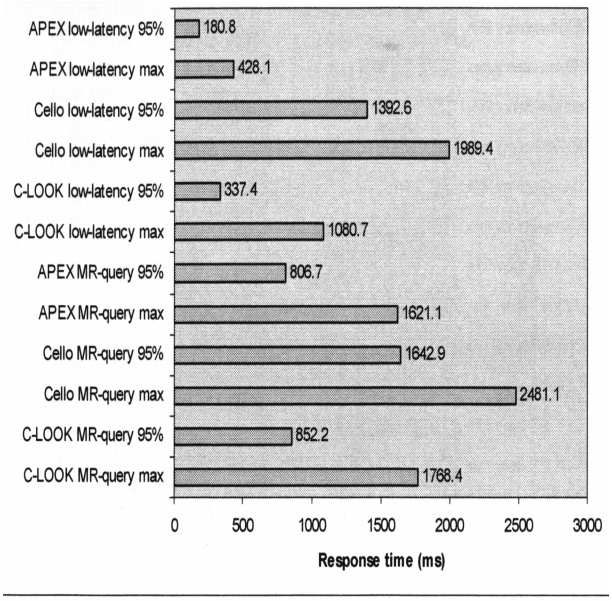


Figur 5.5: Eksperiment 4 - Fordeling av responstider APEX, Cello og C-LOOK (ref [13])

- Arbeidsbevarende skedulering, ved at man legger inn et slakk når man bestemmer størrelsen på et kolli i forhold til rundetiden.
- Dynamisk opprettelse av køer under kjøretid, der køene fjernes når de ikke er i bruk for å redusere overhead.

Vi har også sett at APEX gjennom simuleringer har vist seg å gi god ytelse i mange forskjellige sammenhenger. Den rene ytelsesorienterte skedulereren C-LOOK ga bedre ytelse totalt sett, men brøt tidsfrister og hadde en spredning i sine resultater som ikke er forenelig med et system som skal gi garantier. APEX overholdt tidsfristene konsekvent, imot et ganske lite tap i total ytelse.

I neste kapittelet viser vi hvordan essensen i disse prinsippene kan tilpasses til en implementasjon på operativsystemnivå, og beskriver en praktisk implementasjon.



Figur 5.6: Eksperiment 4: Fordeling av responstider APEX, Cello og C-LOOK (ref [13])

Kapittel 6

Implementasjon av APEX

Dette kapittelet beskriver vår implementasjon av APEX under operativsystemet Linux, og hvilke valg og avgrensninger som er tatt i forhold til den opprinnelige APEX-implementasjonen som beskrives i [13].

6.1 Implementasjonsplatform

Som vi så i kapittel 4 vil en multimediaserver behandle store mengder data. For å skape et realistisk scenario har vi valgt operativsystemet Linux versjon 2.6 som platform. Linux har lenge vært utbredt som serverplattform, og de nye egenskapene til Linux 2.6 gjør det aktuelt å bruke Linux i store installasjoner. Eksempelvis støttes opptil 64 prosessorer, 64Gb RAM og filsystemer med en størrelse på opptil 16Tb.

I versjon 2.6 av Linux er det gjort omfattende omskrivninger av IO-subsystemet i forhold til tidligere versjoner. Disse omskrivningene gjør systemet spesielt egnet for uttesting av nye disk-skedulerere: Systemet kommer med flere innebyggede disk-skedulerere, man kan bytte og velge hvilken skedulerer som skal benyttes for hver enkelt lagringsenhet mens systemet kjører. Dette gjør det enklere å sammenligne ytelsen til APEX med andre IO-skedulerere under like forhold, på samme maskinvare- og software-installasjon. OS-kjernen støtter også at disk-skedulerere kan utvikles og kompileres som en kjernemodul som kan lastes under kjøretid. Ved å implementere APEX som en kjernemodul kan man levere APEX som en separat programvarepakke som kan lastes inn og tas i bruk uten behov for rekompilering eller vesentlig endring av den øvrige kjernekoden. Dette øker mulighetene for praktisk anvendelse av våre resultater.

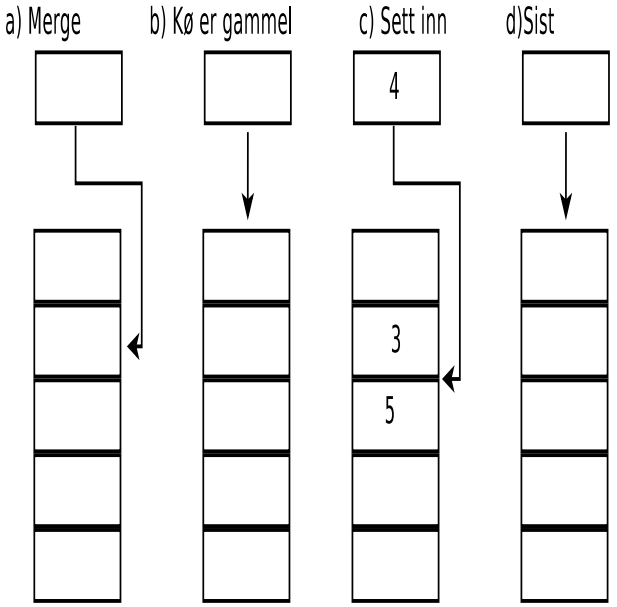
I det følgende vil vi gå igjennom hvordan disk-skedulering i Linux fungerer. Deretter gir vi en beskrivelse av hvilke føringer dette har hatt for vår implementasjon.

6.2 Innebygde diskskeduleringsalgoritmer i Linux 2.6

Linux versjon 2.4 bruker skedulereren Linus elevator, men i kjerneversjon 2.6.10 er Linus elevator supplert av tre nye skedulerere som standard: Deadline IO, Anticipatory IO og CFQ. Man kan konfigurere hvilken skedulerer som benyttes. Her gir vi en oversikt over de algoritmene som allerede er tilgjengelige. For en mer detaljert gjennomgang, se [12].

6.2.1 Linus elevator

I versjon 2.4 av kjernen brukes den såkalte Linus elevator, en enkel kø der man setter inn nye forespørsler etter dette prinsippet:



Figur 6.1: Linus elevator

1. Dersom det kommer inn en forespørsel som gjelder en sektor som ligger ved siden av en sektor som allerede er etterspurt av en forespørsel i køen, slås den nye forespørselen sammen med den eksisterende forespørselen.

2. Hvis de eksisterende forespørslene i køen er for gamle, kommer den nye forespørselen bakerst i køen.

3. Hvis det går an å plassere den nye forespørselen mellom to forespørsler avhengig av sektor på disken, plasseres den der i køen. Gitt at det finnes en forespørsel etter sektor 4 og 6, og det kommer en forespørsel etter 5, da settes den nye forespørselen imellom 4- og 6-forespørselen i køen.

4. Ellers plasseres den nye forespørselen bakerst i køen.

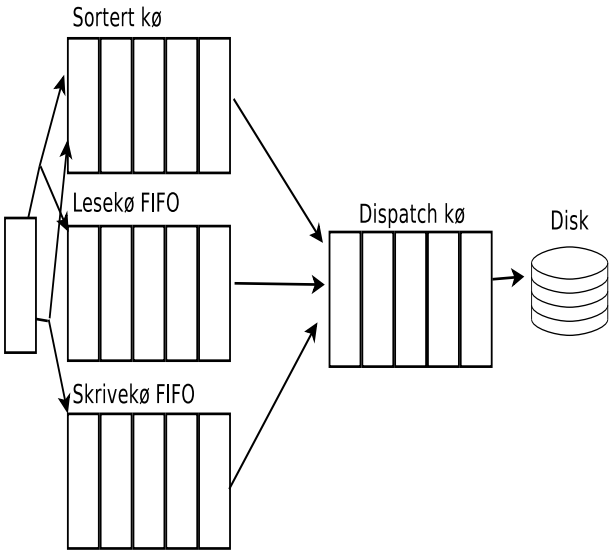
Punkt 2) fungerte dårlig, fordi implementasjonen av mekanismen som sjekker alderen på forespørsler var dårlig. Meningen med den er å hindre utsulting slik at ikke forespørsler skal bli liggende i køen for alltid. Dette ble gjort ved at man gikk over til FIFO dersom det fantes forespørsler som var eldre enn en gitt grense. Å rette på dette problemet dette ble sett på som veldig viktig gjennom utviklingen av 2.4-kjernen.

6.2.2 Deadline IO

Deadline IO forsøker å løse problemet med utsulting og er basert på ideen om at lese-forespørsler er viktigere enn skriveforespørsler. Alle forespørsler har en deadline som de skal behandles innenfor: Lese-forespørsler har 500 millisekunder, skriveforespørsler har 5 sekunder. Det opereres med tre køer:

- En sortert kø som i Linus elevator.
- En FIFO-kø for leseforespørsler
- En FIFO-kø for skriveforespørsler

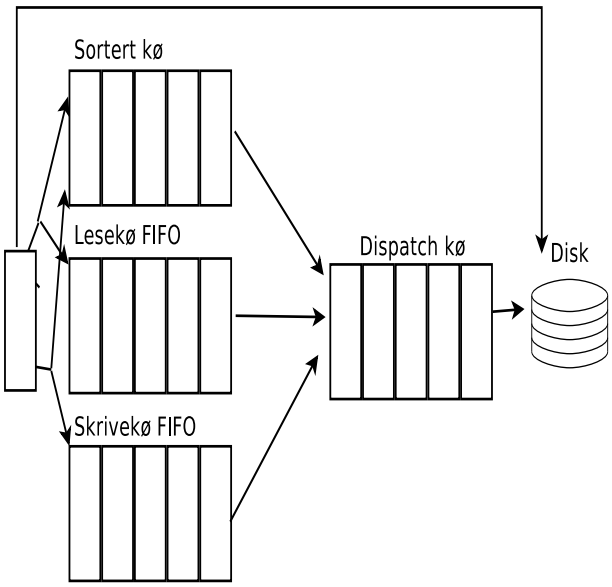
En forespørsel legges både inn i den sorterte køen, og i lese- eller skrivekøen. Normalt behandles forespørselen som ligger i den sorterte køen, men hvis fristen har utløpt for forespørselen først i køen på en av de to FIFO-køene tas neste forespørsel herfra. På denne måten får man fordelene ved en sortert kø kombinert med en ok håndtering av utsultingsproblemet i krisetilfeller.



Figur 6.2: Deadline IO

6.2.3 Anticipatory IO

Anticipatory IO bygger på Deadline IO, og har tre køer og tidsfrister på samme måte. Den forsøker imidlertid å håndtere problemet med at leseforespørsler som regel kommer etter hverandre fra en applikasjon, for eksempel hvis man leser en lang sammenhengende mediafil, men at de kommer med et lite mellomrom i tid, slik at man risikerer å gå frem og tilbake for å håndtere leseforespørsler som egentlig hører sammen. Man har derfor lagt inn en liten pause på et konfigurerbart antall millisekunder etter hver lese-forespørsel, og hvis det dukker opp noen forespørsler fra applikasjonen på tilstøtende områder på disken behandles disse først. Se også [10].



Figur 6.3: Anticipatory

6.2.4 CFQ

Complete Fair Scheduler (CFQ) bygger på Anticipatory IO. I CFQ legges forespørslene i køer avhengig av den kjørende prosessens prosessgruppe, gruppeid eller brukerid. Tiden som hver kø bruker blir registrert, og køene blir sortert etter hvor lang tid de har brukt. På den måten får hver kø en tilnærmet lik andel av tiden for å kjøre.

6.2.5 Timesliced CFQ

Timesliced CFQ er en videreutvikling av CFQ. I timesliced CFQ får hver prosess sin egen kø, og hver kø får eksklusiv adgang til å utføre IO for en tildelt andel av tiden, derav navnet timesliced. Når køen er tom kan den stå uvirksom i påvente av at flere forespørsler skal komme inn innen tiden er omme. Hvorvidt en kø venter på flere forespørsler eller ikke er avhengig av køens prioritet, se under. I de tilfellene der den venter på flere forspørsler får man den samme effekten som i anticipatory IO. Timesliced CFQ støtter tre forskjellige klasser av køer:

Idle Forespørsler som ligger i denne køen blir hentet ut når ingen andre prosesser kjører IO-operasjoner.

Best effort Forespørsler som ligger i denne typen køer blir hentet ut når denne køen har tur. Køen har en prioritet, som avgjør hvor stor tidsandel den får tildelt.

Realtime Sanntidskøene fungerer akkurat som best-effort køen, men køer av denne typen får alltid en del av tiden for hver eneste runde.

Prioriteten til en prosess er et tall mellom 0 og 8, som avgjør hvor stor tidsandel prosessens forespørsler skal motta. Den kan settes eksplisitt, ved at den arves fra foreldreprosessens prioritet, ellers ved at

prosessen får en prioritert basert på sin nice-verdi.

6.2.6 Grensesnitt for bytte av disk-skedulerer

Linux har to viktige egenskaper som har innvirkning på grensesnittet:

- Muligheten til å eksportere egenskaper ved kjernen til filsystemet *sysfs*.
- Dynamisk lasting av kjernekode under kjøring.

Kjernen kan vise frem egenskaper i konfigurasjon og kjøretidsvariable som filer under stien */sys/*. Filene som vises frem kan også være skrivbare, slik at man kan angi parametere til kjernen mens systemet kjører. Konfigurasjon av diskskedulering er gjort på denne måten, ved at man skriver til filer i *sysfs*.

Linux her en monolittisk kjerne, og endringer i kjernen krever i utgangspunktet rekompilering og omstart av systemet. Kjernemoduler gjør det enklere å endre kjernens oppførsel, ved at man kan laste inn kode i kjernen mens systemet kjører. Kjernemoduler er også praktisk motivert av hensynet til distribuert utvikling av kjernekode.

Se A.4 for et eksempel på hvordan man kan laste og bytte disk-skeduleringsmodul, og hvordan man kan hente ut informasjon om denne.

6.2.7 Oppsummering av eksisterende skedulerere

Vi har nå sett hvordan de forskjellige disk-skedulererne i Linux 2.6 fungerer. Fokus for disse skedulererne er høyest mulig gjennomstrømning. Deadline IO forsøker å løse problemet med utsulting som fantes i Linus elevator. Anticipatory IO bringer inn et nytt element ved at den ser diskforespørselene i sammenheng for å øke diskutnyttelsen. CFQ bringer inn et nytt element, ved at den gir de forskjellige prosessene en forholdsmessig del av båndbredden. Timesliced CFQ bringer inn nok et nytt element i forhold til de foregående skedulererne ved at den støtter tjenestekvalitetsparametre for prosesser. Med unntak av timesliced CFQ er det ingen av de andre som støtter forskjellige typer tjenestekvalitet.

6.3 APEX-implementasjonen

Denne seksjonen beskriver vår implementasjon av APEX. For en detaljert teknisk redegjørelse, se A.5.

6.3.1 Avgrensninger i forhold til den opprinnelige APEX

I kapittel 5 ga vi et sammendrag av hovedprinsippene i APEX slik de er beskrevet i [13]. Under implementasjonen av APEX har vi tatt enkelte valg som skiller seg fra den opprinnelige stukturen, men hovedprinsippene er bevart.

Den største forskjellen er at APEX er forutsatt implementert i user-space i forbindelse med en MMDBMS i [13], mens vi har implementert APEX som en generell disk-skedulerer på OS-nivå. Vi har derfor valgt bort en del av de omkringliggende mekanismene i [13], som tilgangskontroll, transaksjonshåndtering og båndbredde-håndterer.

I [13] var tjenestekvalitet knyttet til transaksjoner. I vår implementasjon får hver prosess sin egen kø, og prosessen kan kun velge en type tjenestekvalitet av gangen. Dersom en prosess har flere filer

åpne samtidig vil disse motta samme tjenestekvalitet. Til tross for ulempen dette medfører, forenkler det grensesnittet for å velge tjenestekvalitet i et generelt operativsystem.

I [13] forutsatte man en mekanisme for tilgangskontroll og håndtering av båndbredde. I vår implementasjon legges det ingen begrensninger på hvilke prosesser som kan kjøre, eller hvilke tjenestekvalitetsparametre som kan settes. Når en prosess for eksempel ber om en viss sanntidsgaranti får den det uten at det tas hensyn til trafikken ellers på systemet. Støtten for sanntidstjeneste og tjenestekvalitet gir derfor ikke noen garanti for at kravene skal oppfylles. Et system som tar helhetlige hensyn om hva som er mulig å oppnå er tenkelig, men faller utenfor denne oppgaven.

I [13] ble det forutsatt at hver runde skulle ha en fast lengde. I vår implementasjon starter vi en ny runde når det er mindre enn to forespørsler igjen fra forrige runde som skal ferdigbehandles. Dersom det finnes sanntidskøer som venter med forespørsler bestemmes lengden av runden av den tidligste tidsfristen til køen, ellers begrenses rundetiden av den estimerte tiden det tar å behandle 20 forespørsler. Vi får dermed variable rundetider i vår implementasjon.

I [13] ble det forutsatt at man hadde en egen tjenestetype for lav forsinkelse, og at forespørsler av denne typen skulle sendes til driveren i midten av en runde. Vi har ikke implementert denne tjenestetypen.

I [13] går man igjennom alle køene to ganger: Først en gang hvor man tar hensyn til tokens, og så en gang til i en arbeidsbevarende fase, hvor man plukker forespørsler uavhengig av køenes tokenverdier helt til kolliet er fylt opp. Her er det også slik at best-effort køene ikke har tokenverdier, og er avhengige av den arbeidsbevarende fasen for å få kjørt sine forespørler. I vår implementasjon har også best-effort-køene tokenverdier, og vi har kuttet ut den arbeidsbevarende fasen.

6.3.2 APEX basert på CFQ

Vi har valgt å ta utgangspunkt i CFQ-skedulereren under implementasjonen av APEX, da dette er den skedulereren som ligger nærmest opp til APEX.

Her er en oversikt over forskjeller og likheter mellom APEX og CFQ:

- Både APEX og CFQ oppretter og fjerner køer dynamisk.
- Begge skedulererne er inspirert av anticipatory IO, og er arbeidsbevarende ved at de utnytter slakk.
- APEX legger opp til et mer fingradert system for tjenestekvalitet enn CFQ.
- APEX støtter kø med sanntidsgaranti, CFQ gjør ikke dette foreløbig.
- APEX bruker utvidet token bucket for å bestemme hvilke forespørsler som skal behandles, mens CFQ bruker gir en andel av tiden til hver kø.

Å basere implementasjonen på CFQ gir visse føringer for hvordan de forskjellige delene i APEX er implementert. Eksempelvis er kollibyggeren laget slik at den kalles opp hver gang en forespørsel ankommer systemet, og hver gang diskdriveren spør om en ny forespørsel. I [13] forutsatte man kollibyggeren kjører i en egen tråd.

I Linux 2.6 er kildekoden for disk-skedulerere organisert på en spesiell måte slik at det skal være enkelt å bytte skedulerer under kjøretid. Dette rammeverket legger også en del begrensninger på hvordan koden i vår implementasjon er organisert. Se appendiks for en mer detaljert oversikt over hvordan rammeverket med de forskjellige funksjonene til en disk-skedulerer er bygget opp.

6.3.3 Viktige konsekvenser av avgrensningene

Mangelen på en mekanisme som avviser tjenestekvalitetskrav når systemet ikke gir dekning for å oppfylle dem, innebærer at kravene vil bli brutt når arbeidslasten blir stor nok. Denne implementasjonen kan altså ikke garantere at tjenestekvalitetskravene blir oppfylt, den vil kun forsøke å oppfylle dem etter beste evne.

Når denne implementasjonen lar tjenestekvalitetskrav være knyttet til prosesser istedenfor å knytte dem til transaksjoner, fører det til at en prosess bare kan motta en type tjenestekvalitet av gangen. Dette gjør systemet mindre anvendelig i situasjoner der prosesser har mange åpne filer samtidig, og i situasjoner der applikasjoner har sammensatte krav som bør ses i forhold til hverandre, som transaksjoner. På den annen side blir grensesnittet for å sette tjenestekvalitetskrav enklere.

I denne implementasjonen bruker vi tiden det tar å behandle tyve forespørsler som rundetid, og denne tiden kan reduseres dersom det er sanntidsforespørsler med en tidligere tidsfrist. I [13] ble det imidlertid forutsatt at rundene skulle ha en fast lengde. Konsekvensen av dette valget er at vi får variabel lengde på rundene i vår implementasjon. Når det bygges flere kolli innenfor en runde, vil disse begrenses av størrelsen på runden. Når rundelengden varierer vil størrelsen på koliene også variere. Ved svært korte rundelengder vil kolliene som bygges bli mindre.

Den arbeidsbevarende fasen hvor køene gås igjennom uten hensyn til tokenverdier er sentral i [13]. I denne implementasjonen har vi likevel valgt å la best-effort-køene ha tokenverdier, og droppet den arbeidsbevarende fasen. Dette har trolig innvirkning på størrelsen på de kolliene som bygges, de blir mindre når man ikke har med en arbeidsbevarende fase som kjøres uten hensyn til tokenverdier.

En egen tjeneste for lav forsinkelse er ønskelig i mange sammenhenger når brukeren interagerer med en applikasjon. Dette var også en viktig del av APEX slik det er beskrevet i [13]. Denne implementasjonen støtter ikke slik tjeneste for lav forsinkelse, noe som gjør implementasjonen mindre anvendelig i et realistisk scenario.

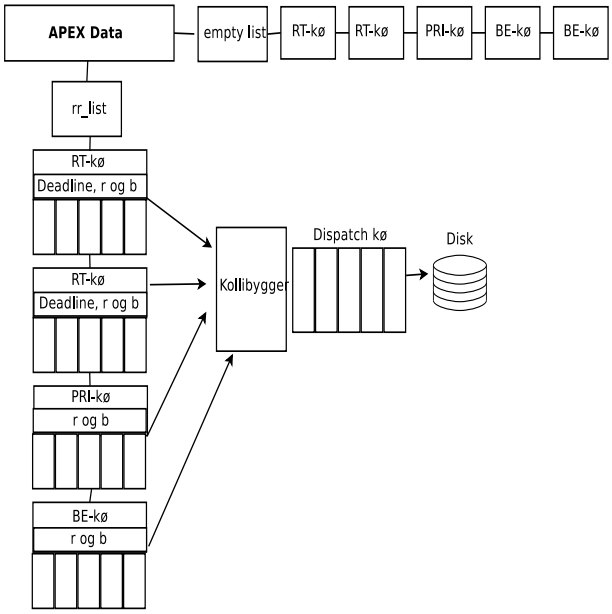
6.3.4 Oversikt over implementasjonen

Figur 6.4 viser datastruktur og den overordnede datastrømmen i APEX. Skedulereren har en datastruktur *apex_data* som inneholder to lister med køer: *empty_list* som inneholder køer som ikke har forestående IO-operasjoner i øyeblikket, og *rr_list*, som inneholder køer som har IO-operasjoner som har forespørsler som venter på å bli utført. Når en ny forespørsel ankommer, plasseres den først i den kjørende prosessens kø, og deretter flyttes køen fra *empty_list* til *rr_list*.

Den sentrale delen av APEX er kollibyggeren, som plukker ut forespørsler fra de forskjellige køene og sender dem videre til disken som en samlet enhet. Forespørslene i køene som ligger i *rr_list* er sortert etter FIFO-prinsippet innad, altså i den rekkefølgen de kom inn. Kollibyggeren velger ut forespørsler avhengig av hvilken klasse køen tilhører, og hvilke attributter den har, og sender dem videre i sortert rekkefølge til dispatch-køen, hvor disk-driveren overtar.

I vår implementasjon av APEX får hver prosess sin egen kø. Køene kan tilhøre en av tre forskjelige mulige klasser: Sanntid (RT), Prioritet (PR) eller Best-Effort (BE). Alle køene har parameterne *r* og *b*. *r* angir hvor mange tokens køen får tildelt i sekundet, mens *b* angir hvor mange tokens køen kan samle opp. Ved å regulere verdien av *b* kan man unngå at en kø får en uforholdsmessig stor andel av båndbredden i forhold til de andre køene ved plutselige utbrudd av mange forespørsler til denne køen.

En sanntidskø har en tidsfrist for hvor lang tid som kan gå før forespørsler i denne køen skal være



Figur 6.4: APEX datastruktur

ferdigbehandlet. Forespørsler i disse køene blir behandlet først, og tidsfristen kan føre til at forespørseler i disse køene blir behandlet selv om køen ikke har tokens.

En prioritetskø får bare sendt over forespørsler til kollibyggeren dersom det finnes tokens i køen.

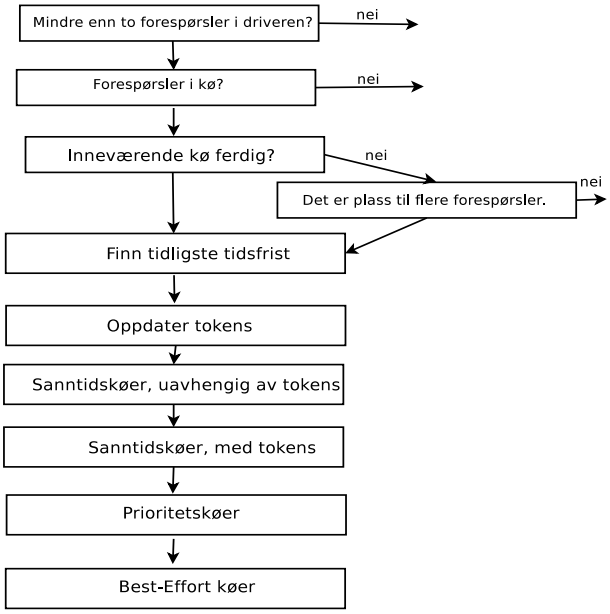
Forespørsler i best-effort-køene får bare kjørt dersom køen har tokens og dersom det er plass i inneværende runde etter at kollibyggeren har gått igjennom sanntidskøene og prioritetskøene.

6.3.5 Kollibyggeren

Flytdiagrammet i figur 6.5 viser hvordan kollibyggeren fungerer. Tidsbegrepet er sentralt i APEX. Vår implementasjon vedlikeholder et estimat, *tes*, for hvor lang tid en diskforespørsel vil ta. Estimatet oppdateres hele tiden avhengig av hvor lang tid diskforespørslene faktisk tar. Estimatet er konservativt: Dersom en forespørsel tar mindre tid enn *tes*, reduseres estimatet med et millisekund. Dersom en forespørsel tar lengre tid enn *tes* økes estimatet med ti millisekunder. Vi forsøkte opprinnelig å justere estimatet opp og ned med ett millisekund, men fant det nødvendig å vekte justeringen i favør av forespørsler som tar lang tid for å hindre at estimatet ble for lite. Ti millisekunder er derfor valgt på bakgrunn av empiriske tester.

Når et kolli bygges

Rutinen som kan bygge et nytt kolli kjøres hver gang driveren spør etter en ny diskforespørsel, og hver gang en ny diskforespørsel ankommer APEX. Et kolli bygges kun når det er færre enn to ubehandlede forespørsler igjen fra forrige kolli. Dersom det fortsatt er en forespørsel igjen i driveren når rutinen kjøres, brukes den inneværende rundens slutt for å avgjøre hvor mange forespørsler som skal sendes avgårde. Dersom det ikke er noen ubehandlede forespørsler igjen i driveren, bygger man et nytt kolli



Figur 6.5: Kollibyggerens arbeidsflyt

til en ny runde. Først finner man ut når den nye runden skal slutte. Dette er tiden det tar å foreta tyve forespørsler, eller den tidligste tidsfristen blant sanntidsforespørslene. Til slutt sendes det avgårde så mange forespørsler som det er plass til innenfor denne tiden.

Tidligste tidsfrist

Dersom forrige runde er forbi, antar vi i utgangspunktet at en runde skal inneholde tyve forespørsler. Tidspunktet for slutten av runden blir derfor

$tstart + 20tes$

Deretter sjekkes sanntidskøene for å se om noen av dem har en tidligere tidsfrist, og isåfall justeres tiden for avslutning av runden tilsvarende. Størrelsen på kolliet blir dermed:

tstart	Tidspunkt for begynnelsen av runden.
tes	Estimat for hvor lang tid en forespørsel tar.
ted	Tidspunkt for når runden slutter.
n	Nåværende tidspunkt.
dr	Antall forespørsler i driveren.
tokens	Antall tokens en kø har.
since_last_update	Tidsrom siden forrige token-oppdatering.
r	Antall tokens en kø mottar i sekundet.
b	Maksimalt antall oppsamlede tokens en kø kan ha.

Tabell 6.1: Forkortelser brukt i beskrivelsen av APEX

$floor(\frac{ted-tstart}{tes})$

Der *ted* er tidspunkt for slutten av runden, *tstart* er tidspunktet for begynnelsen av runden, og *tes* er estimatet for hvor lang tid en diskforespørsel tar.

Antall forespørsler som kan legges inn beregnes slik:

$floor(\frac{ted-n}{tes}) - dr$

Der *n* er nåværende tidspunkt, og *dr* er antall forespørsler som allerede ligger i driveren.

Oppdatering av tokens

Når kollibyggeren vet hvor mange forespørsler som skal velges ut, kjøres rutinen for å oppdatere antall tokens for prioritetskøene. Som vi så over har disse køene en verdi *r* og en verdi *b*, der *r* er antall tokens køen mottar i sekundet, og *b* er det maksimale antall tokens som køen kan samle opp. Hver kø holder regnskap med når siste oppdatering ble foretatt, og utregningen blir dermed:

$tokens + floor(\frac{since\_last\_update*r}{1000})$

Der *tokens* er det antallet tokens køen har fra før, og *since_last_update* er antall millisekunder siden køen sist fikk oppdatert sine tokens. Dersom resultatet av denne utregningen blir større enn *b* settes antall tokens for denne køen til *b*.

Utvelgelse fra køene

Når kollibyggeren vet hvor mange forespørsler det er plass til i neste kolli, og køene har fått oppdatert sine tokenverdier, går den først igjennom sanntidskøene som har forestående forespørsler, og velger ut de forespørslene som ligger i disse køene som må plukkes ut for å overholde tidsfrister. Deretter går kollibyggeren igjennom sanntidskøen på nytt, denne gangen for å velge forespørsler etter hvor mange tokens disse køene har. Dersom det er plass til flere, velges det ut forespørsler fra prioritetskøene etter hvor mange tokens disse køene har. Hvis det er plass til enda flere forespørsler i kolliet etter at prioritets-køene har blitt behandlet, velger kollibyggeren ut det restendende antallet forespørsler fra køene som er av typen best effort, avgrenset oppad til antallet tokens disse køene har samlet opp.

Dersom ingen køer har tokens, slik at ingen forespørsler er valgt når vi er ferdig med best-effort køen, har vi et spesialtilfelle der det er kapasitet på disken, og for lite tokens til køene. I såfall foretas en ny runde gjennom alle køene, og det velges ut en forespørsel fra hver kø, avgrenset oppad av størrelsen på kolliet som bygges.

6.3.6 Grensesnitt

Som vi så over får hver prosess sin egen kø. Køen kan sette tjenestekvalitetsparametre ved å skrive til spesielle filer i *sys-fs*. Disse filene ligger i
/sys/block/<diskpartisjon>/queue/iosched/:

- *apex_queuetype* Kan settes til RT (Sanntid), PR (Prioritet) eller BE (Best effort).
- *apex_r* Angir raten av tokens som tildeles denne køen per sekund.
- *apex_b* Angir maksimalt antall tokens denne køen kan samle opp.

- *apex_deadline* Angir tidsfrist i antall millisekunder for hvor lang tid som kan gå før en forespørsel er ferdigbehandlet.

En applikasjon kan skrive til disse filene for å angi ønsket verdi for disse parameterne. Alle køtypene kan ta *r* og *b* som parameter, men det er bare sanntidskøer som tar *deadline*-parameteret. Dersom man ikke angir noen parametre blir køen en best effort-kø, med *r* satt til 10 og *b* lik 50.

6.4 Oppsummering

I dette kapittelet har vi har sett hvordan disk-skedulering fungerer i Linux 2.6. Vi har også beskrevet hvordan vi har implementert diskskeduleringsalgoritmen i APEX innenfor disse rammene, med CFQ som utgangspunkt. I neste kapittel gjennomgår vi tester av denne implementasjonen, og sammenligner disse resultatene med noen av de øvrige disk-skedulererne i Linux 2.6.

Kapittel 7

Test og testresultater fra APEX-implementasjonen

I dette kapittelet tester vi APEX-implementasjonen og sammenligner den med skedulereren CFQ og de andre skedulerne i Linux 2.6. Først behandles testmiljø og arbeidslast som er valgt ut for å kjøre testene. Til sist går vi igjennom testresultatene og analyserer disse.

7.1 Testmiljø

Maskinen som er brukt i testene kjører Linux 2.6.10 har en AMD Athlon 1.3 prosessor, og 750Mb RAM. Disken som brukes i testene er en 120Gb Western Digital Caviar SE EIDE 120 GB 7,200 RPM EIDE med følgende spesifikasjoner:

Egenskap	*Verdi*
Rotational Speed	7,200 RPM (nominal)
Buffer Size	8 MB
Average Latency	4.20 ms (nominal)
Contact Start/Stop Cycles	50,000 minimum
Seek Times (Average)	
Read Seek Time (Average)	8.9 ms
Write Seek Time (Average)	10.9 ms (average)
Track-To-Track Seek Time	2.0 ms (average)
Full Stroke Seek	21.0 ms (average)

7.2 Arbeidslast

For å få en realistisk arbeidslast bruker vi tyve forskjellige ti-minutters videoer som leses med en rate på 288KB i sekundet, og to forskjellige store filer på 9.2Gb som leses sekvensielt så raskt som mulig, se figur 7.1. Vi benytter en serie Python-script for å simulere klientene som leser filene. Testene er kjørt med en reboot mellom hver test, for å unngå at resultatet skal påvirkes av at operativsystemet mellomlagrer de blokkene som allerede er lest.

Figur 7.1: Arbeidslast med lesning av 22 forskjellige filer

7.3 Testresultater

Først gjennomgås eksperimentet og de tilpasningene som er gjort til APEX og CFQ for å foreta de forskjellige målingene. Deretter går vi igjennom resultatene fra de forskjellige målingene og sammenligner de forskjellige skedulererne. Til slutt analyserer vi resultatene.

7.3.1 Eksperimentet

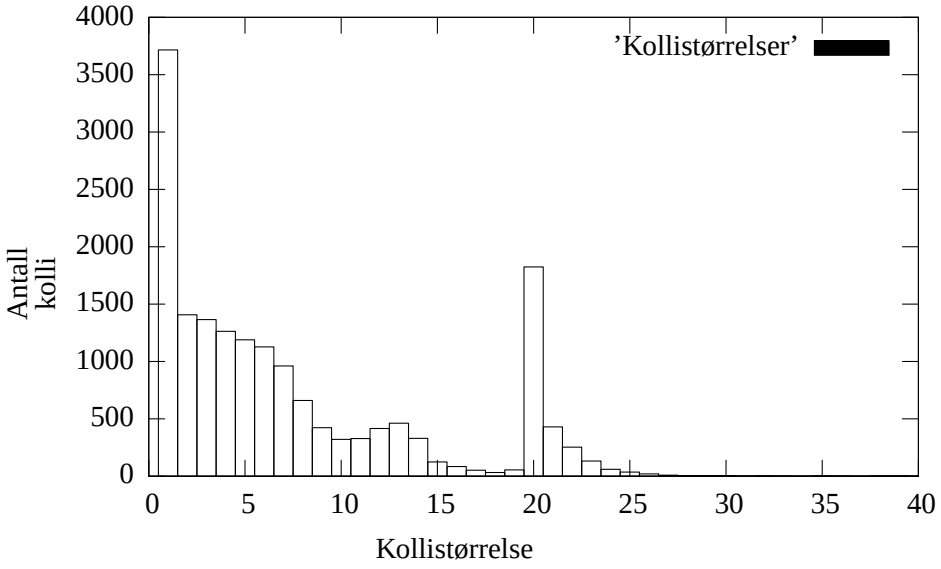
To klienter startes opp som leser fra de to store filene så raskt som mulig i ti minutter. Dersom disse klientene når slutten på filen, begynner de fra begynnelsen igjen. Disse to klientene representerer "bakgrunnsstøy" i målingen, og vi teller hvor mange 64K-blokker disse prosessene klarer å lese i løpet av ti minutter. Disse leser som best-effort, og får en rate *r* på 100 tokens i sekundet, maksimalt 1000 akkumulerte tokens.

Vi starter opp ti sanntidsklienter som leser hver sin ti minutters video med en tidsfrist på 200ms for hver forespørsel. Disse får en rate *r* på 5 tokens i sekundet, og maksimalt 100 akkumulerte tokens. Videoene leses med en rate på 288KB/sekundet.

Vi starter ti prioritetsklienter som leser hver sin ti minutters video med en rate *r* på 5 tokens i sekundet, maksimalt 100 akkumulerte tokens. Disse videoene leses med en rate på 288KB/sekundet.

7.3.2 Tilpasninger for måling av APEX-implementasjonen

For å måle egenskapene ved APEX-implementasjonen har vi lagt inn logging på to forskjellige steder. Vi benytter mekanismer som allerede er innebygget: Funksjonen *apex_enqueue* kjøres hver gang en forespørsel ankommer skedulereren, og knytter tidspunktet for starten på behandlingen til forespørselen. I *apex_account_completion* brukes dette tidspunktet for å logge tiden det tar fra en forespørsel ankommer



Figur 7.2: Fordeling av kollistørrelser i APEX

og til den er ferdigbehandlet. Denne funksjonen kjøres hver gang en forespørsel er ferdigbehandlet. Det er den globale variabelen *jiffies* som brukes for å måle tid i kjernen. Vi logger altså den sammenlagte tiden forespørselen bruker i kø før den blir behandlet, og tiden disken bruker på å behandle den. I funksjonen *apex_dispatch_requests*, hvor kollibyggingen foregår, logger vi antall forespørsler som har plass i hvert kolli som bygges, og hvor mange forespørsler som faktisk sendes ut i hvert kolli.

7.3.3 Tilpasninger for måling av CFQ

For å måle responstidene for forespørsler i CFQ på samme måte som i APEX skrev vi om diskskedulereren slik at den gir en kø til hver prosess, som i APEX. Responstiden måles og logges i *cfq_enqueue* og *cfq_account_competion*, på samme måte som i APEX.

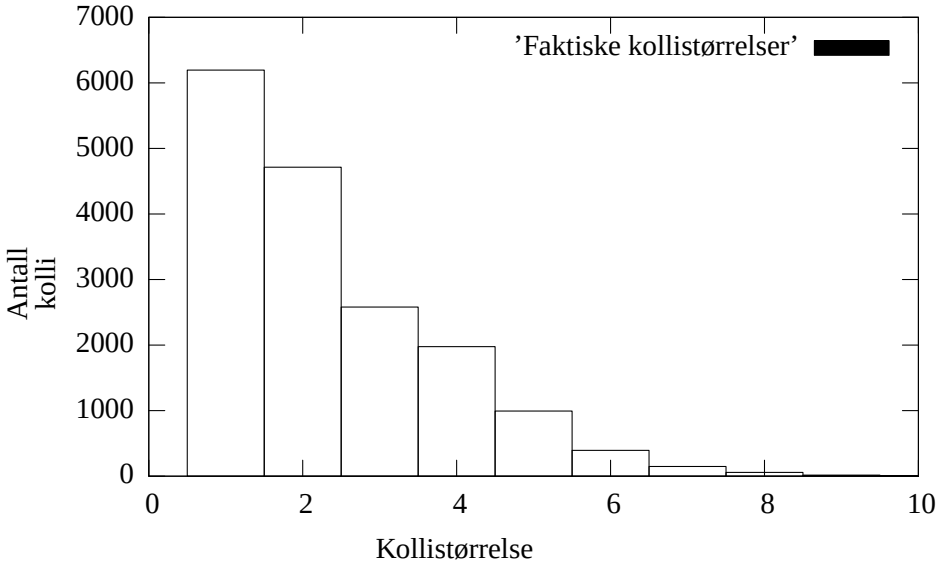
7.3.4 Presentasjon og sammenligning av resultater

APEX

For APEX-skedulereren la vi inn logging av størrelsen på kolliene som ble bygget. Dette er APEX-spesifikt, og viser store variasjoner i rundene.

Figur 7.2 viser distribusjonen av størrelsen på kolli, altså hvor mange forespørsler som kan plukkes ut hver gang et kolli bygges. Det er svært mange kolli som kun består av en enkelt forespørsel, så avtar størrelsene gradvis ned til en liten topp på 12-13 forespørsler, og så er det en stor andel kolli som 20 forespørsler. Det er også noen få kolli med størrelse over 20.

Som vi så i forrige kapittel er 20 standardstørrelsen når et nytt kolli bygges dersom det ikke er noen tidsfrister som gjør at vi må lage et kolli som består av færre forespørsler enn 20. Dette forklarer den store toppen på 20. Vi husker også at vi bygget et nytt kolli dersom alle forespørslene i det inneværende



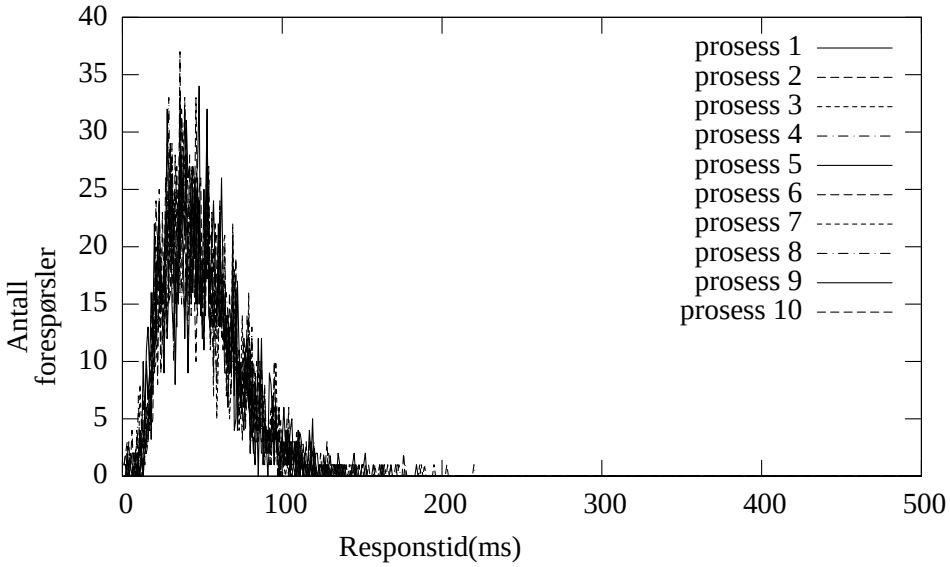
Figur 7.3: Fordeling av faktisk størrelse på kolli i APEX

kolliet var ferdigbehandlet før tiden. Dette forklarer at det er relativt få kolli like under 20. Den lille toppen rundt 12-13 forespørsler kan tyde på at det er mange kolli som er ferdig etter at 7-8 forespørsler er blitt behandlet. En stor andel av kolliene består av fem eller færre forespørsler. Dette kan skyldes at tidsfristene gjør at det kun er plass til noen forespørsler i kolliet. Til sist har vi kolliene som har en størrelse på over 20. Dette virker pussig, siden 20 er standardstørrelsen som settes når det ikke er noen tidsfrister som begrenser kolliet. Dette kan tilskrives variasjoner i tidsanslaget for hvor lang tid en forespørsel tar, *tes*. Dersom man setter isammen et kolli på 20 med en høy *tes*-verdi, kan denne gjøre seg ferdig før tiden samtidig som *tes* synker, slik at det blir plass til flere enn 20 forespørsler innenfor den opprinnelige tidsfristen.

Figur 7.3 viser fordelingen av de faktiske størrelsene på kolli som ble satt sammen. Vi ser at antallet forespørsler som faktisk sendes ut som et kolli er vesentlig mindre enn det som er mulig ifølge figur 7.2. Den største toppen ligger på kollistørrelsen en, dernest to, og så forekommer det i gradvis mindre og mindre grad størrelser opp til 8.

Hovedinntrykket fra denne grafen er at kollistørrelsene er mye mindre enn det man kunne forvente. Det er altså ikke nok forespørsler tilgjengelig som står i kø når et kolli bygges, selv med to prosesser som leser kontinuerlig, og 20 prosesser som leser med en bitrate på 288KB/sekundet. Det er også interessant å se denne grafen i sammenheng med kollistørrelsene i figur 7.2. Kolliene som faktisk bygges har en maksimal størrelse på 8, og vi ser av figur 7.2 at kollistørrelsene har en topp på 7-8 forespørsler under 20. Dette kan skyldes at kolliene som har plass til 20 forespørsler blir ferdigbehandlet etter 7-8 forespørsler, og så bygges det nye kolli innenfor den samme fristen. Det store antallet forespørsler på en og to stemmer godt med det vi så i figur 7.2.

Figur 7.4 viser responstidene for sanntidsprosessene i APEX. X-aksen viser hvor mange millisekunder som brukes for hver forespørsel, mens Y-aksen viser antall forespørsler som hadde denne responstiden. De ti prosessene som kjører som sanntid er tegnet inn. Det er vanskelig å skille de ti prosessene



Figur 7.4: APEX responstider, sanntidskøer

fra hverandre ut ifra denne grafen, men det er store forskjeller innad mellom de ti prosessene, men vi har en tydelig topp rundt 40-70 ms. Ellers holder alle prosessene seg innenfor sanntidsgarantien på 200ms, bortsett fra noen få forespørsler som ligger opp mot 220ms.

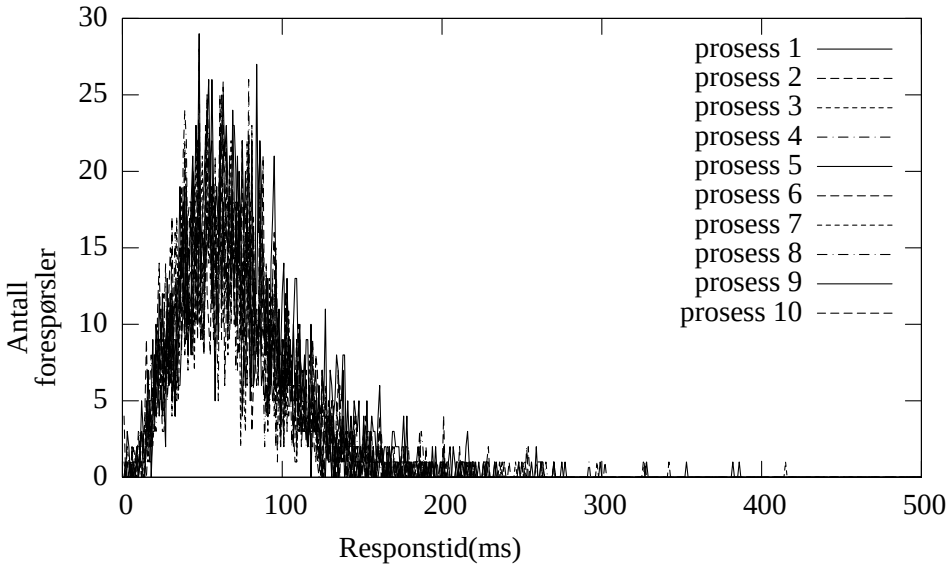
En ting som er verdt å merke seg ved denne grafen er at noen forespørsler hadde en behandlingstid som lå over grensen på 200ms. Dette kan skyldes at det var mange forespørsler til behandling akkurat da denne forespørselen ankom. Vår APEX-implementasjon gir ingen garantier om at tjenestekvalitets-parameterne overholdes. Den overveiende delen av forespørslene holdt seg imidlertid innenfor 200ms-fristen.

Figur 7.5 viser responstidene for prioritetsprosessene i APEX. X-aksen viser antall millisekunder per forespørsel, mens Y-aksen viser antall forespørsler med den aktuelle behandlingstiden. Disse prosessene kjører uten tidsfrist, men med samme *r* og *b*-verdi som sanntidsprosessene. Også her er det vanskelige å skille prosessene fra hverandre ut i fra grafen, men som i tilfellet med sanntidsprosessene er det store forskjeller innad imellom prosessene, med en klar topp mellom 40 og 70ms.

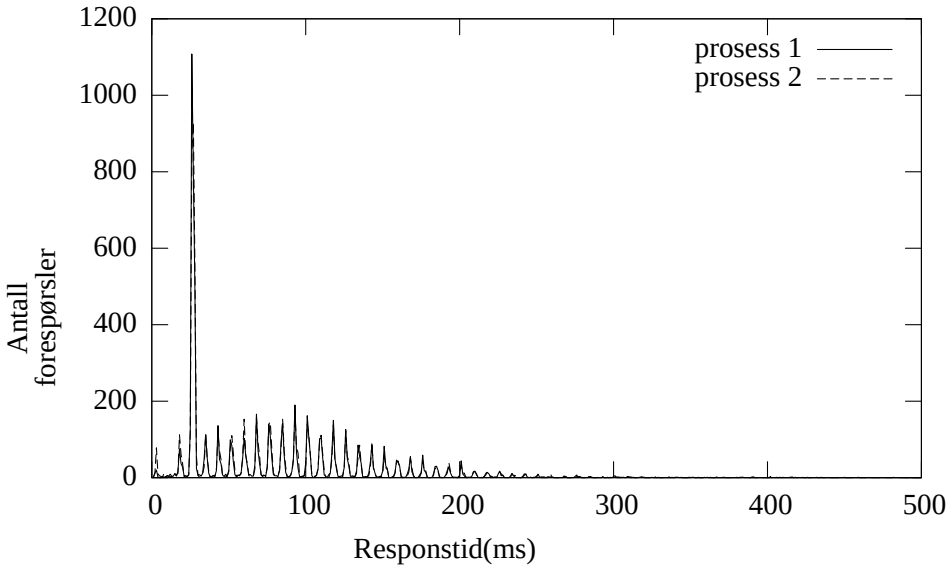
Det er interessant å se denne grafen i forhold til grafen over sanntidsprosessene, her er det en klart større andel av forespørslene som tar over 100ms enn det som var tilfellet for sanntidsforespørslene. Vi ser også har prioritetsprosessene har langt flere forespørsler som bruker lenger tid enn sanntids-prosessenes tidsfrist på 200ms. Det ser også ut til at det her er større variasjon innad mellom prosessene.

Figur 7.6 viser responstidene for best-effort-prosessene i APEX. Disse to prosessene leser uten opphold fra store filer så fort de kan. X-aksen viser antall millisekunder per forespørsel, mens Y-aksen viser antall forespørsler med den aktuelle behandlingstiden. Her er det en klar topp rundt 27ms, etterfulgt av flere små topper med avtagende høyde. De tregeste forespørslene ligger på rundt 280ms.

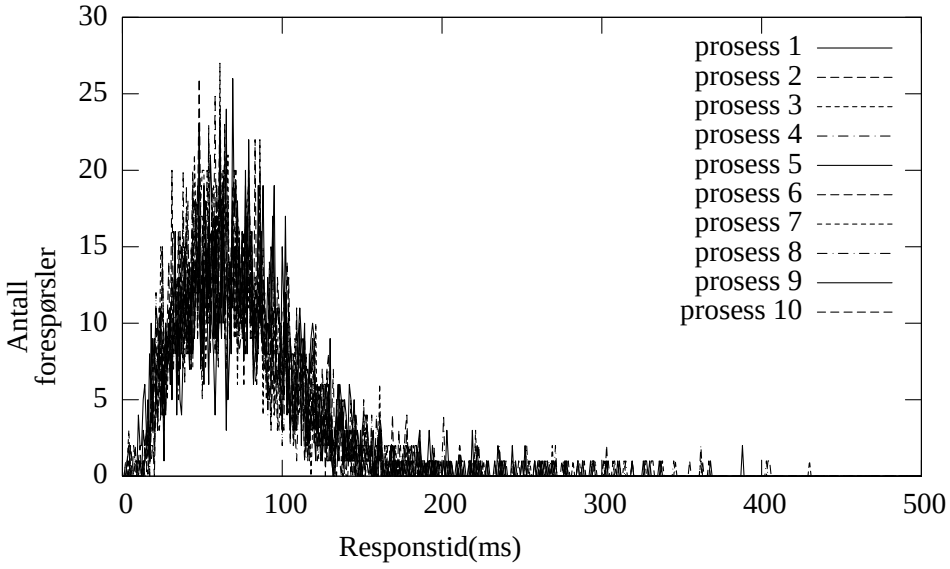
De to prosessene følger nesten nøyaktig samme bølgemønster, med periodevise topper og “bølge-daler”.



Figur 7.5: APEX responstider, prioritetskøer



Figur 7.6: APEX responstider, best effort



Figur 7.7: CFQ responstider, sanntidskøer

CFQ

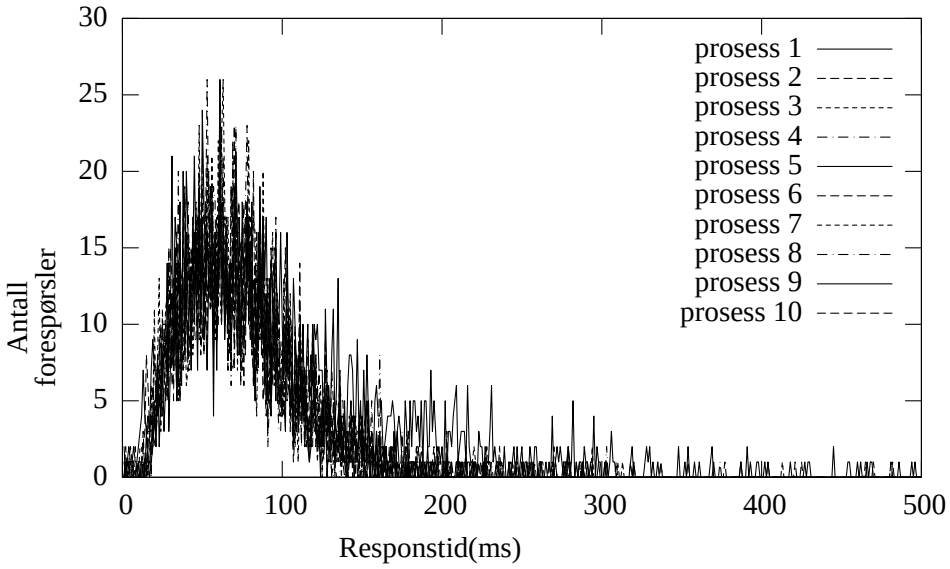
Figur 7.7 viser responstidene for de ti første prosessene i CFQ. Også her viser X-aksen antall millisekunder, mens Y-aksen viser antall forespørsler med den aktuelle responstiden. Prosessene leser med en rate på 288KB/sekundet, men her har vi ingen tidsfrist eller *r* og *b*-verdier. Det er vanskelig å skille prosessene fra hverandre, men vi har en tydelig topp rundt 50ms, og mange prosesser som ligger over APEX-prosessenes grense på 200ms. De tregeste forespørslene har en behandlingstid på 400ms. Det er en del variasjoner i behandlingstid innad mellom disse prosessene.

Figur 7.8 viser responstidene for de neste ti prosessene i CFQ. Disse opererer under like forhold som i forrige grafen. X-aksen viser antall millisekunder per forespørsel, Y-aksen viser antall forespørsler med det aktuelle antall millisekunder. Også her er det vanskelig å skille prosessene fra hverandre, men vi ser at de tregeste forespørslene behandlingstider opp mot 500ms. Det er en del forskjeller innad imellom prosessene.

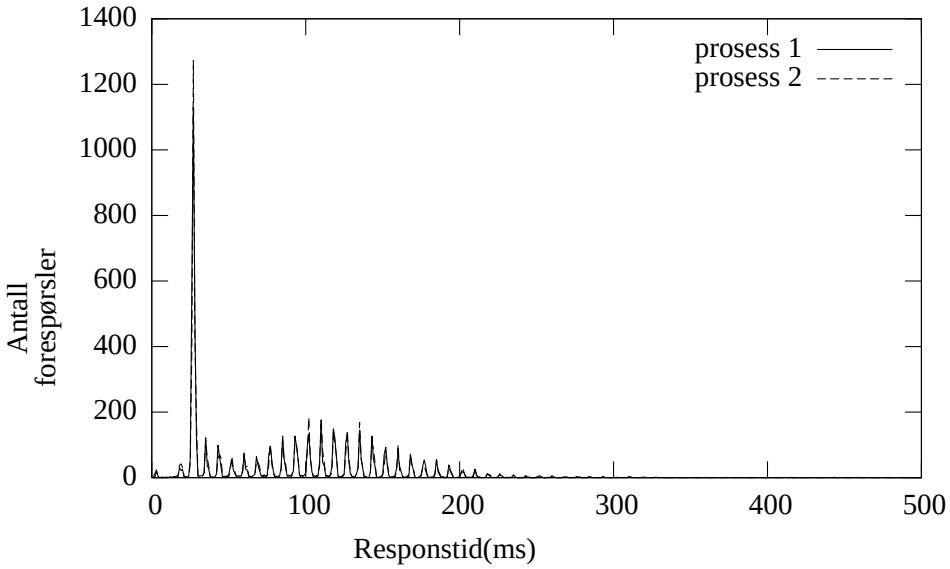
Formen på denne grafen er lik grafen over de første ti prosessene, slik man kunne forvente siden de kjører under like forhold. Det er imidlertid interessant å se hvor mange av disse prosessene som bruker godt over 200ms på å behandles.

Figur 7.9 viser responstidene for best-effort-prosessene i CFQ. Disse prosessene leser kontinuerlig fra to store filer. X-aksen viser antall millisekunder, Y-aksen viser antall forespørsler med det aktuelle antall millisekunder. Denne grafen følger samme bølgemønster som vi så i APEX sine best-effort prosesser. Vi har en klar topp rundt 27, så gradvis større topper opp mot 100ms. Deretter følger periodevist avtagende topper. De tregeste prosessene ligger på rundt 450ms.

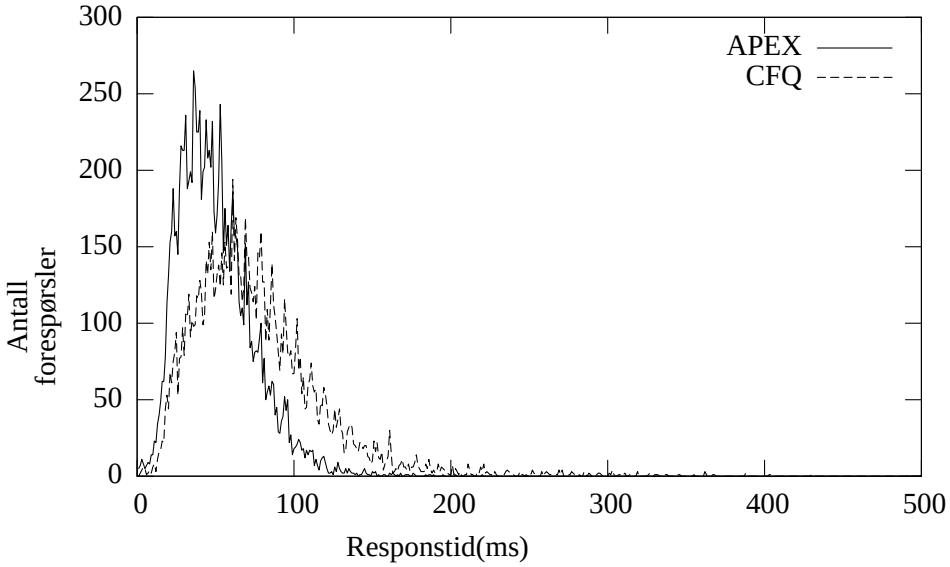
Det er interessant å se hvordan disse to prosessene følger nesten nøyaktig samme form.



Figur 7.8: CFQ responstider, prioritetskøer



Figur 7.9: CFQ Responstider, de to best-effort prosessene



Figur 7.10: Responstider, sanntidskøer

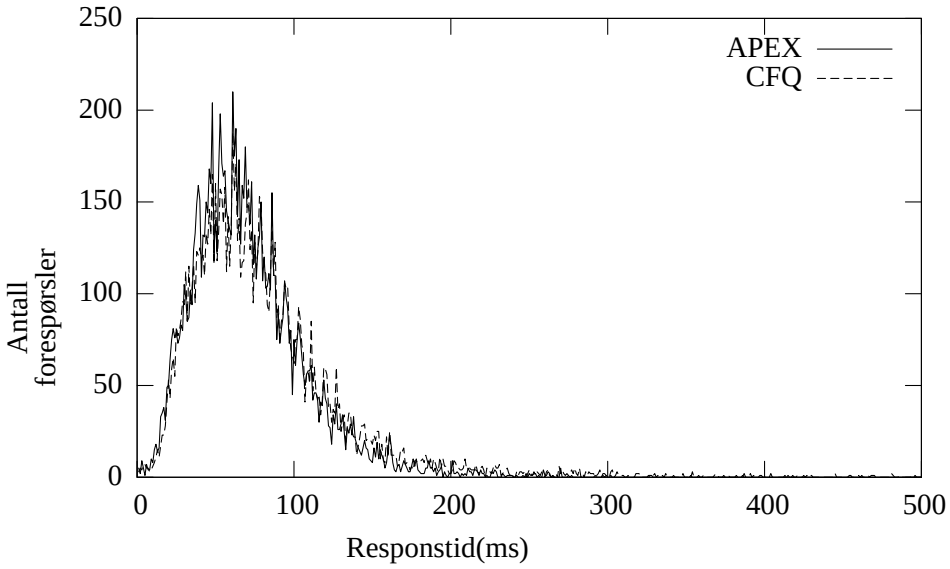
APEX versus CFQ

I figur 7.10 har vi lagt sammen responstidene til de ti sanntidsprosessene til APEX, og de ti første prosessene til CFQ. Også her viser X-aksen responstiden i millisekunder, mens Y-aksen viser det aktuelle antallet forespørsler med det aktuelle antall millisekunder. Vi ser at APEX-prosessene har en større andel forespørsler som tar mindre enn 70ms, mens CFQ-prosessene har et større antall forespørsler som tar mer enn 70ms.

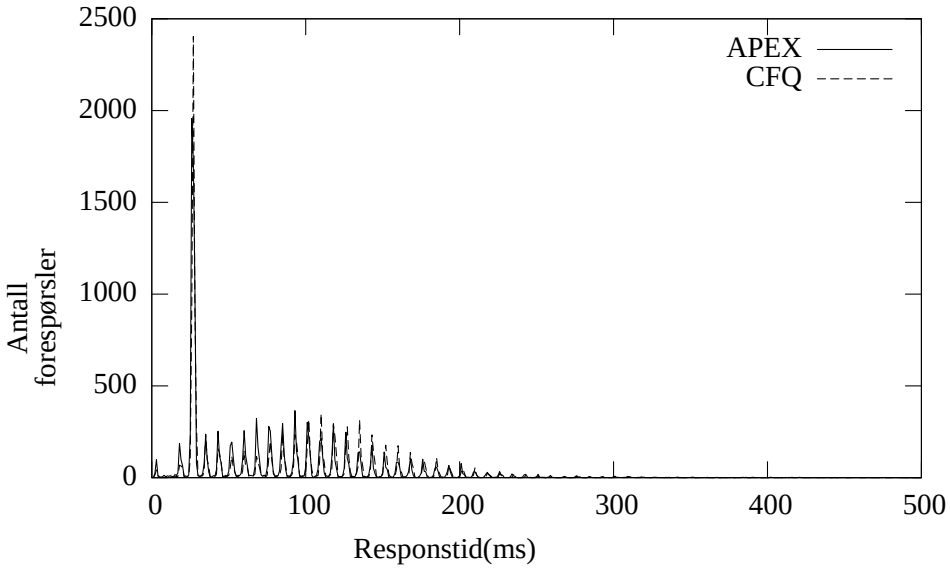
Her ser vi at sanntidsmekanismen i APEX har en klar effekt når det gjelder å begrense behandlingstiden for sanntidsforespørsler. APEX-prosessene overholder tidsfristen sin har med noen få unntak, og majoriteten av forespørslene holder seg innen tidsfristen med god margin.

I figur 7.11 har vi lagt sammen responstidene til de ti prioritetprosessene i APEX, og de ti neste prosessene i CFQ. Både APEX- og CFQ-prosessene leser med en rate på 288KB/sekundet. Begge prosessgruppene har samme form, og deretter en avtagende tendens helt frem til de tregeste prosessene, som har en behandlingstid opp mot 500ms. Det er bare små forskjeller som skiller disse prosessgruppene fra hverandre.

I figur 7.12 har vi lagt isammen responstidene til de to best-effort prosessene i APEX, og de to prosessene som leser store filer i CFQ. Både APEX-prosessene og CFQ-prosessene leser store filer så raskt de kan. X-aksen viser responstiden i antall millisekunder, mens Y-aksen viser antall forespørsler med det aktuelle antallet millisekunder. Både APEX-prosessene og CFQ-prosessene følger samme form, med en stor topp på 27ms fulgt av gradvis stigende topper mot 100ms, deretter gradvis avtagende topper frem mot 280ms.



Figur 7.11: Responstider, prioritetskøer



Figur 7.12: Responstider, best effort

7.3.5 APEX versus andre skedulerere

I dette avsnittet sammenligner vi responstidene for APEX med de andre skedulererne. Deretter sammenligner vi gjennomstrømningen for best-effort prosessene i APEX og de andre skedulererne.

Sammenligning av responstider

Kildekoden for de andre skedulererne er organisert slik at det er vanskelig å foreta målinger i kjernen slik vi har gjort for APEX og CFQ. Vi har isteden foretatt målinger i user-space av tiden det tar å lese en 64K-blokk fra disken, og sammenlignet disse responstidene for APEX med resten av skedulererne. Disse testene er implementert i C, og C-programmet benytter RDTSC-instruksjonen for å måle tid. Denne instruksjonen finnes på Intel-prosessorer, og returnerer en *long long int* som teller klokkesykler. For å regne ut tidsbruk dividerer man antall klokkesykler som har gått med antall klokkesykler maskinen bruker per sekund. Koden som måler tidsbruken for en leseoperasjon ser slik ut:

```
read_start = get_rdtsc();
chars_read = fread(buf, 1, 64000, fp);
read_stop = get_rdtsc();
```

Slike målinger er unøyaktige: For det første måler vi en operasjon som kanskje gjennomføres i form av flere faktiske forespørsler til disken, og kjøretidsmiljøet kan velge å lese mer enn det som trengs slik at innholdet på disken blir mellomlagret i minnet. For det andre får vi med tid som brukes før leseoperasjonen havner i køen til skedulereren, og tiden som går med etter at forespørselen er ferdig i skedulereren. Når man måler i user-space er det også slik at man får med tid som brukes av andre prosesser dersom det skjer et kontekstbytte i løpet av perioden man måler.

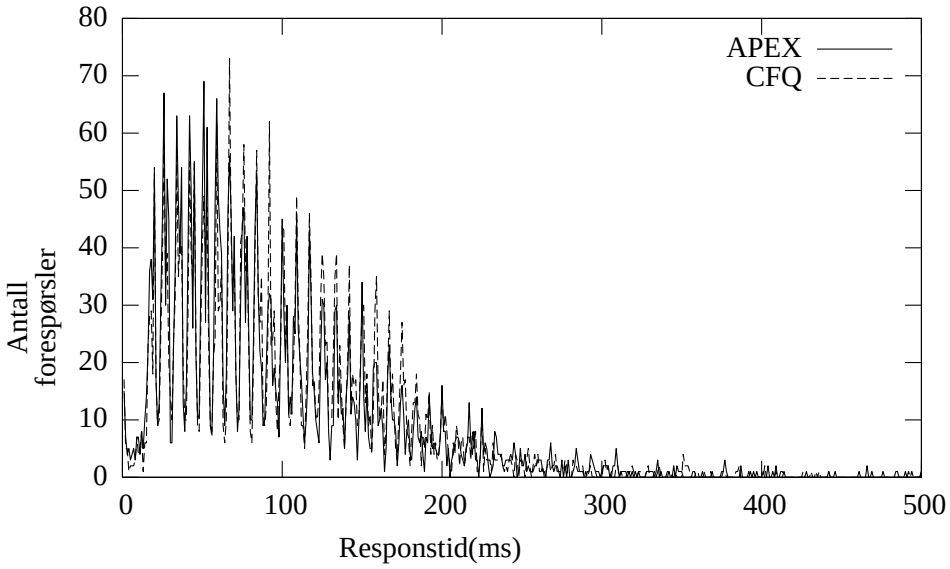
Under disse målingene fant vi at de fleste lese-operasjonene tok mindre enn et millisekund å behandle. Dette bekrefter at det leses mer enn 64K hver gang, og at resultatet av operasjonen mellomlagres i minnet. I det følgende har vi valgt å ignorere de operasjonene som tok 0ms, og viser kun forespørsler som tok målbar tid.

Vi starter med figur 7.13, som viser fordelingen av responstidene til de ti sanntidsprosessene til APEX, sammen med de tilsvarende ti første prosessene til CFQ. X-aksen viser responstiden i millisekunder, mens Y-aksen viser antall forespørsler med den aktuelle responstiden. Her har vi allerede foretatt en nøyaktig måling i kjernen, se figur 7.10. I denne nye målingen har begge skedulererne responstider på opp mot 500, og det er langt større variasjoner innad for de to prosessgruppene. Vi kan likevel se at APEX har en større andel av forespørsler som tar under 70ms.

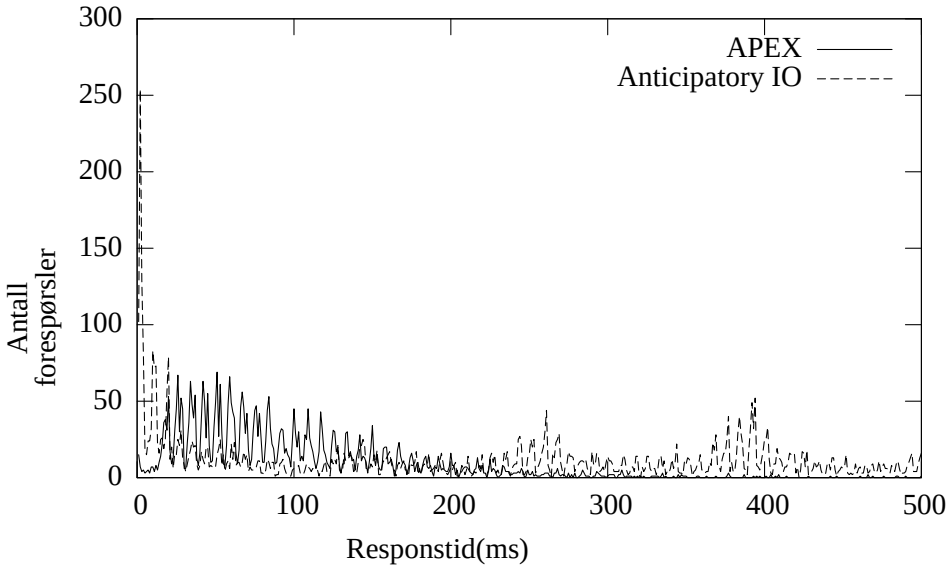
I figur 7.14 er fordelingen av responstider for de ti sanntidsprosessene til APEX tegnet inn sammen med de tilsvarende ti første prosessene i Anticipatory IO. X-aksen viser responstiden i millisekunder, mens Y-aksen viser antall forespørsler med den aktuelle responstiden. Anticipatory IO har et stort antall forespørsler som tar ett millisekund, og en mye større andel av forespørsler som tar over 200ms.

Den høye andelen av forespørsler på ett millisekund skyldes at allerede innleste data mellomlagres av operativsystemet.

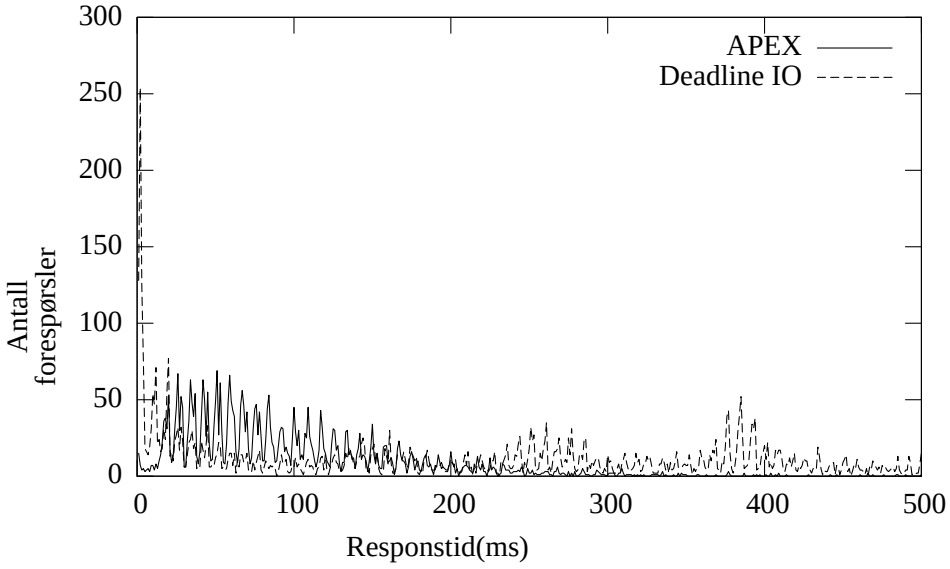
I figur 7.15 er fordelingen av responstider for de ti sanntidsprosessene til APEX tegnet inn sammen med de tilsvarende ti første prosessene i Deadline IO. Også her viser X-aksen responstid i millisekunder, mens Y-aksen viser antall forespørsler med den aktuelle responstiden. Som med Anticipatory IO har Deadline IO et høyt antall forespørsler som tar 1ms, fulgt av to store topper på henholdsvis 250ms og



Figur 7.13: APEX versus CFQ, sanntidskøer målt i user-space



Figur 7.14: APEX versus Anticipatory IO, sanntidskøer målt i user-space



Figur 7.15: APEX versus Deadline IO, sanntidskøer målt i user-space

380ms. Her ser vi tydelig hvordan de fleste av APEX-prosessene skiller seg fra Deadline IO fordi de fleste responstidene ligger under 200ms.

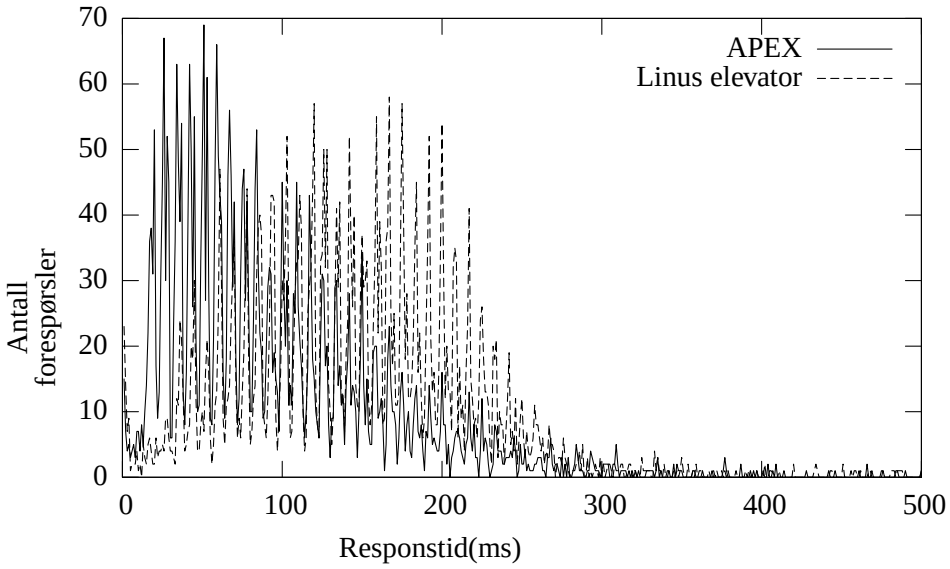
Også her ser vi at vi har en stor andel forespørsler med responstid på ett millisekund, som betyr at operativsystemet har mellomlagret allerede innleste data.

I figur 7.16 ser vi igjen fordelingen av responstidene for de ti sanntidsprosessene til APEX. Her er også de tilsvarende ti første prosessene til Linus Elevator tegnet inn. Linus Elevator har en klart større andel av forespørsler som tar over 100ms enn det APEX har.

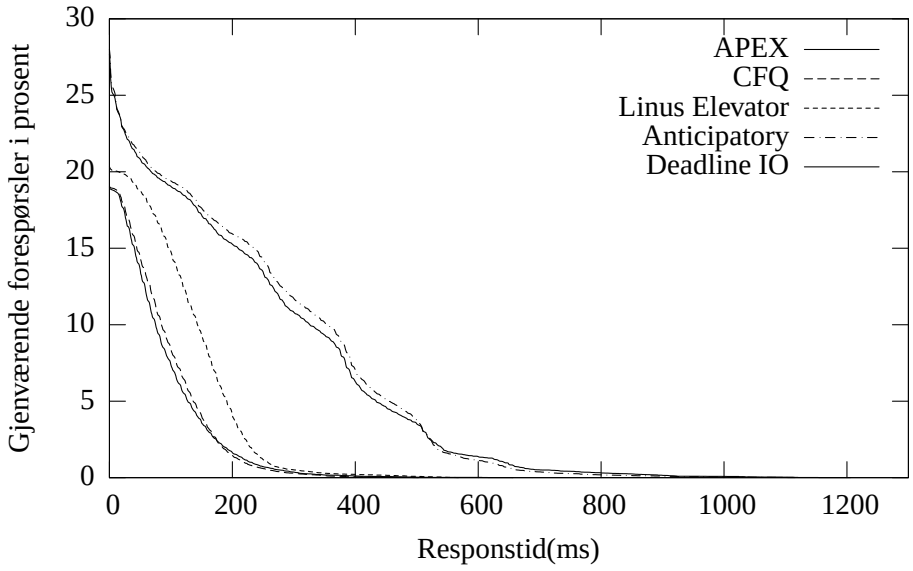
Figur 7.17 viser hvor stor andel av forespørslene som er ferdig etter et gitt antall millisekunder. APEX, CFQ, Anticipatory IO, Deadline IO og Linus Elevator er tegnet inn. X-aksen viser antall millisekunder, Y-aksen viser hvor stor prosentandel av forespørslene som gjenstår etter et gitt antall millisekunder. Det er bare 18-27 prosent av forespørslene som tar målbar tid, noe som tyder på at leseoperasjonen fører til at fler blokker enn nødvendig blir lest for hver gang, og så mellomlagres disse slik at neste leseoperasjon ikke går til disken, men til operativsystemets buffer-cache.

Figur 7.18 viser det samme som figur 7.17, men her er Y-aksen logaritmisk, slik at man kan se mer detaljert som skiller de forskjellige skedulererne. APEX, CFQ, Anticipatory IO, Deadline IO og Linus Elevator er tegnet inn. X-aksen viser antall millisekunder, og den logaritmiske Y-aksen viser hvor stor prosentandel av forespørslene til hver prosessgruppe som gjenstår etter et gitt antall millisekunder.

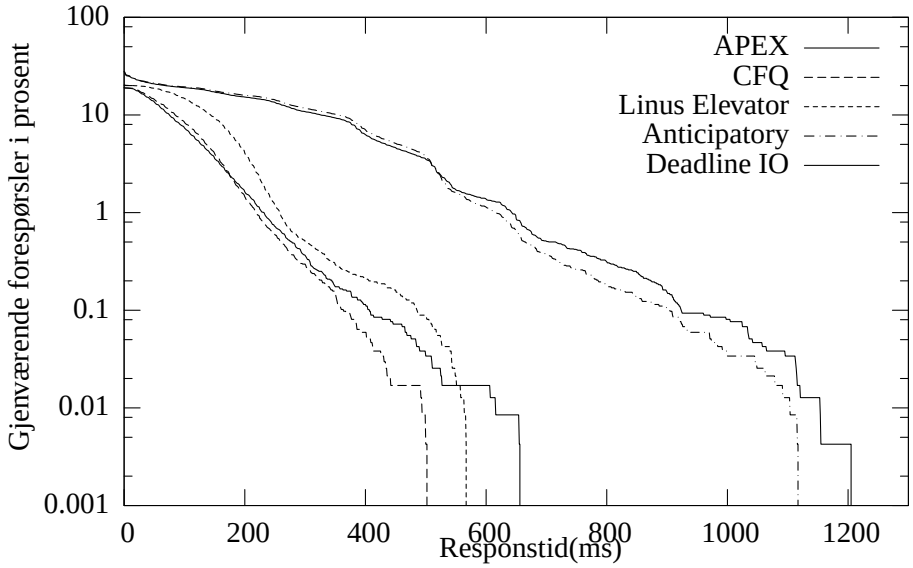
APEX har lavest responstid for rundt 97% av forespørslene, deretter har den noen få forespørsler som går forbi CFQ og noen enda færre forespørsler som er tregere enn Linus elevator. Den tregeste forespørselen er på over 650ms. CFQ har høyere responstid enn APEX for rundt 97% av forespørslene, men har raskere responstid enn de andre skedulererne. Den tregeste forespørselen er på rundt 500ms. Linus elevator har høyere responstid enn APEX og CFQ for rundt 99.9% av sine forespørsler. Den tregeste forespørselen er på 580ms. Anticipatory IO og Deadline IO har høyere responstid enn alle de andre skedulererne. Den tregeste forespørselen til Anticipatory IO er på over 1100ms, mens Deadline IO



Figur 7.16: APEX versus Linus Elevator, sanntidskøer målt i user-space



Figur 7.17: APEX sanntidskøer versus andre skedulerere, målt i user-space



Figur 7.18: APEX sanntidskøer versus andre skedulerere med logaritmisk Y-akse, målt i user-space

Skedulerer	Første fil	Andre fil
Deadline IO	19182	18647
APEX	16874	17033
Linus Elevator	15055	15077
CFQ	14402	14207

Tabell 7.1: Antall leste blokker for forskjellige skedulerere

har en forespørsel som tar over 1200ms.

APEX har altså den laveste responstiden for de aller fleste forespørslene, selv om vi ser at den har enkeltforespørsler som tar lengre tid enn både CFQ og Linus Elevator. Alle skedulererne har forespørsler som går langt over tidsfristen på 200ms når man måler i user-space. Disse målingene er imidlertid mindre pålitelige enn de sammenligningene vi gjorde over, hvor vi målte responstidene til CFQ og APEX i kjernen.

Sammenligning av gjennomstrømning

Vi har målt gjennomstrømningen for best-effort prosessene på alle skedulererne for å sammenligne. Som vi husker hadde vi to prosesser som sto og leste så mange 64K-blokker som mulig fra to store filer under alle målingene. Tabell 7.1 viser antall blokker som ble lest av hver av disse prosessene.

Som vi ser klarte APEX-prosessene å lese 2-3000 flere blokker enn det CFQ-prosessene klarte. Når disse prosessene når enden på sine 9.2Gb-filer, starter de fra begynnelsen igjen. CFQ har dårligst resultat, med henholdsvis 14402 og 14207 blokker. Deretter følger Linus Elevator med 15055/15077 blokker, etterfulgt av APEX, som klarte 16874/17033 blokker. Deadline IO klarte 19182/1864 blokker.

Vi ser at APEX gir god ytelse i forhold til CFQ i en sammenheng der flere etterfølgende blokker leses etter hverandre. Dette stemmer godt med det vi kunne forvente, siden APEX bruker kolli-prinsippet hvor flere forespørsler sendes over til driveren av gangen.

Resultatene for Anticipatory IO var overraskende, og uforholdsmessig høyt, siden den leste henholdsvis 89629 og 91065 blokker med sine to prosesser. Vi antar at dette skyldes en feil i testscriptet, eller at det skyldes at operativsystemets mellomlagrer allerede leste blokker når prosessen leser filen fra begynnelsen igjen. Det virker imidlertid trygt å anta at Anticipatory IO gir noe bedre ytelse under disse forholdene enn de andre skedulererne, selv om vi ikke kan si noe sikkert om hvor stor denne marginen er.

7.4 Analyse

Gjenom disse testene har vi sett følgende:

- APEX lager kolli som er mindre enn forventet.

  Som vi så i figur 7.2 og figur 7.3 er det stor forskjell på mulig kollistørrelse, og størrelsen på kolliet som faktisk blir satt sammen. Dette kan skyldes at det ikke er mange nok forespørsler tilgjengelig når kolliet settes sammen.

- APEX sin sanntidstjeneste har effekt.

  Som vi så av målingene til sanntidsprosessene i figur 7.4 og i sammenligningen av disse med CFQ-prosessene i 7.10, så har prosessene med sanntidstjeneste vesentlig lavere behandlingstid enn vanlige prosesser, og holder seg stort sett innenfor den fristen som blir satt.

- Sanntidsgarantien er ikke absolutt.

  Denne implementasjonen av APEX gir ingen garanti om at tjenestekvalitetsparameterne som settes kan oppfylles, den gir kun et løfte om at systemet skal forsøke å overholde disse parameterne så godt som mulig. Som vi så i figur 7.4 ble tidsfristen brutt for noen få sanntidsforespørsler.

- APEX gir god ytelse for best-effort køer.

  Opptelling av antall leste blokker for best-effort køene viste at APEX ikke står tilbake for CFQ når det gjelder ytelse for vanlige prosesser. Her skulle man kanskje tro at den mer komplekse behandlingen av forespørslene i APEX ville medføre en overhead som ga seg utslag i dårligere ytelse. Dette er altså ikke tilfelle.

7.5 Oppsummering

I dette kapittelet har vi sett at sanntidstjenesten i denne implementasjonen av APEX har en effekt. Vi har også sett at fristen blir brutt i noen få tilfeller fordi systemet ikke har tilgangskontroll knyttet hvilke tjenestekvalitetskrav som kan stilles. Vi har også sett at APEX-implementasjonen gir like god eller bedre ytelse for sine best-effort køer som andre skedulerere når de opererer med samme arbeidslast. I neste kapittel foretar vi en kritisk analyse av disse resultatene og gir forslag til videre arbeid.

Kapittel 8

Oppsummering og forslag til videre arbeid

I dette kapittelet oppsummerer vi resultatene fra forrige kapittel og foretar vi en kritisk analyse av resultatene i forhold til virkemidlene som er brukt. Til slutt gir vi forslag til videre arbeid.

8.1 Oppsummering og kritisk analyse

I denne oppgaven har vi implementert APEX, en disk-skedulerer som har støtte for tjenestekvalitet, i et virkelig operativsystem. Vi har også sammenlignet denne skedulereren med fire andre disk-skedulerere på samme hardware- og softwareplatform.

Vi har sett at sanntidstjenesten i denne implementasjonen gir vesentlig reduksjon i behandlingstiden for forespørsler når man fastsetter en tidsfrist, og at tidsfristen overholdes i de aller fleste tilfellene. Vi har også sett at denne garantien kan bli brutt dersom man har høy last. Dette skyldes at implementasjonen mangler en mekanisme som overvåker tilgangskontroll og båndbreddehåndtering, systemet forsøker å overholde alle forespørsler om tjenestekvalitet som mottas, og har ikke noen mekanisme for å begrense eller avvise kravene som stilles underveis. Resultatene er positive i den forstand at de viser hvordan mekanismer for å støtte tjenestekvalitet kan bygges, men de illustrerer også viktigheten av en overordnet mekanisme for å gi tilgang til å sette slike parametre.

Ytelsesmessig er disken er en av flaskehalsene i et operativsystem, og prosessorkraft er relativt billig i forhold til den relativt sett lange behandlingstiden disk-forespørsler har. Denne oppgaven bygger på en forutsetning om at det er verdt å bruke prosessorkraft på å organisere disk-skedulering på en slik måte at man oppnår en balanse mellom høy ytelse og ønskelig tjenestekvalitet. Behandlingen av forespørsler er relativt kompleks i forhold til det man finner i andre skedulerere i Linux. Testene av best-effort køer side om side med køer med tjenestekvalitetsparametre viste ikke noe vesentlig ytelsestap som følge av innførselen av tjenestekvalitetsparametre. APEX gjorde det bedre på best-effort trafikk enn både CFQ og Linus elevator. Dette viser at man kan øke kompleksiteten i disk-skedulereren for å innføre tjeneste-kvalitet uten vesentlig tap i ytelse, og er således et positivt resultat.

For å dra nytte av kolli-prinsippet er det ønskelig å samle opp og sende så mange forespørsler som mulig til disken. I testene vi har utført så vi at APEX setter sammen kolli med en størrelse på opp til 8 forespørsler av gangen. Dette er mindre enn forventet, og en konfigurasjon som ga større kolli ville vært å foretrekke. I denne implementasjonen har vi valgt å la rundetidene være variable, og å la den maksimale rundetiden være avhengig av *tes* * 20 forespørsler, istedenfor å sette en fast maksimal rundetid, slik

det ble forutsatt i [13]. Dette valget kan diskuteres, og det kan hende man ville kunnet oppnå større kollistørrelser med en fast rundetid, slik at man kunne vente lenger for å samle opp flere forespørsler.

8.2 Forslag til videre arbeid

Målsetningen for vår implementasjon har vært å demonstrere at APEX kan implementeres i et virkelig operativsystem, for å demonstrere et "proof of concept". En slik demonstrasjon er ikke det samme som å lage et system som kan tas i bruk og settes i produksjon. Under arbeidet med denne oppgaven har vi støtt på enkelte problemstillinger som det ville være interessant å utforske i forbindelse med videre arbeid.

Under testene så vi at kollistørrelsen var ganske liten i forhold til å utnytte kolli-prinsippet fullt ut. En videreutvikling av denne implementasjonen bør forsøke å vente lenger mellom hver runde for å samle opp flere forespørsler. For å oppnå dette bør man trolig innføre en arbeidsbevarende fase hvor forespørsler plukkes ut uavhengig av tokens, slik det er forutsatt i [13]. Man bør trolig også bruke en fast maksimal rundetid istedenfor å la rundetidene variere med tiden det tar å behandle tyve forespørsler.

Vår APEX-implementasjon mangler støtte for en lav-forsinkelse tjeneste, som er nyttig i mange sammenhenger der brukeren interagerer med applikasjoner. Dette er en mangel ved vår implementasjon, og noe som ville gjort systemet mer anvendelig i et realistisk scenario.

Sanntidsstøtten i vår APEX-implementasjon mangler en tilgangskontroll som avviser forespørsler om båndbredde som systemet ikke kan håndtere. Et tillegg med slik støtte ville gjøre systemet mer anvendelig i et realistisk scenario.

I et multimediasystem vil en prossess ofte ha flere filer åpne samtidig, og man kan tenke seg behov for forskjellige tjenestekvalitetsparametre for hver enkelt av de åpne filene. I vår implementasjon får hver prosess en egen kø, og tjenestekvalitetsparameterne som gis for en prosess gjelder for alle filene denne prosessen har på APEX-partisjonen. En implementasjon kunne operere med en tabell med fildeskriptor og prosessid som nøkkel for køene. På den annen side ville dette medføre et mer komplekst grensesnitt for å sette parametere enn det vi har i dag.

Vi har valgt å eksponere grensesnittet for å sette tjenestekvalitetsparametre i *sys-fs*, hvor hver enkelt prosess kan skrive til sine egne parameterfiler. Dette fungerer godt for å demonstrere at det fungerer. I et virkelig system vil det være ønskelig at en prosess kan sette parametre for andre prosesser. For dette formålet vil det trolig være bedre å innføre systemkall til kjernen, istedenfor å bruke *sys-fs*.

Tillegg A

Appendix

A.1 Rammeverk for disk-skedulering i Linux

Kildekoden i Linux som omhandler disk-skedulering er organisert i en fellesdel med et rammeverk som definerer og kaller et standard grensesnitt for diskskeduleringsalgoritmer, og de enkelte diskskedulererne som implementerer dette grensesnittet. Denne seksjonen beskriver dette rammeverket slik det foreligger i Linux 2.6.10.

A.1.1 Grensesnitt

Figuren A.1 viser hvordan grensesnittfunksjonene brukes etterhvert som en forespørsel går igjennom systemet.

Grensesnittet for oppførselen til disk-skeduleringsalgoritmer er definert i */include/linux/elevator.h*, i structen *elevator_ops*. Denne structen har felter for funksjoner av visse typer, og et diskskeduleringsobjekt inneholder en slik struct, fylt med funksjonspekere med samme signatur. På denne måten kan man kontrollere hvordan disk-skedulereren skal behandle forespørsler.

I koden til disk-skedulere er det en konvensjon at disse signaturfunksjonene navngis gis samme navn som grensesnittfunksjonene, bortsett fra at *elevator* erstattes med navnet på disk-skedulereren. I anticipatory scheduler kalles for eksempel *elevator_remove_request* for *as_remove_request*.

Her er funksjonene som er med i *elevator_ops*:

int elevator_merge_fn (request_queue_t *, struct request **, struct bio *)

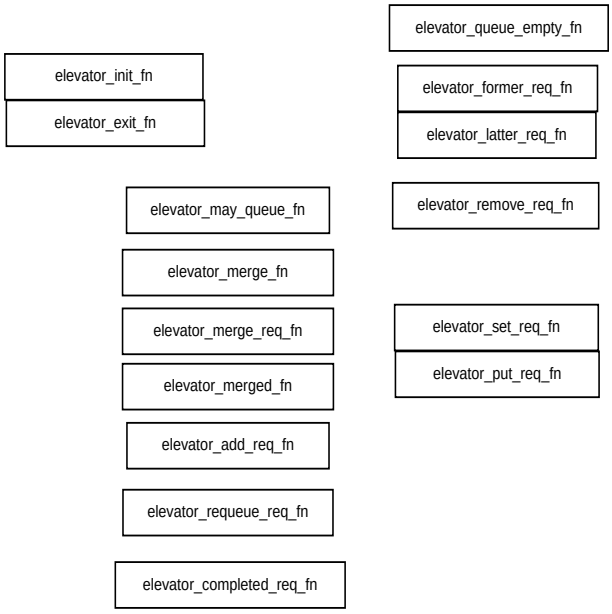
Denne kalles for å avgjøre om det går an å slå sammen en forespørsel om et buffer med en forespørsel som allerede ligger i køen, returner status ELEVATOR_NO_MERGE, eller resultat av elv_try_merge().

Funksjonen er implementert i alle disk-skedulerere.

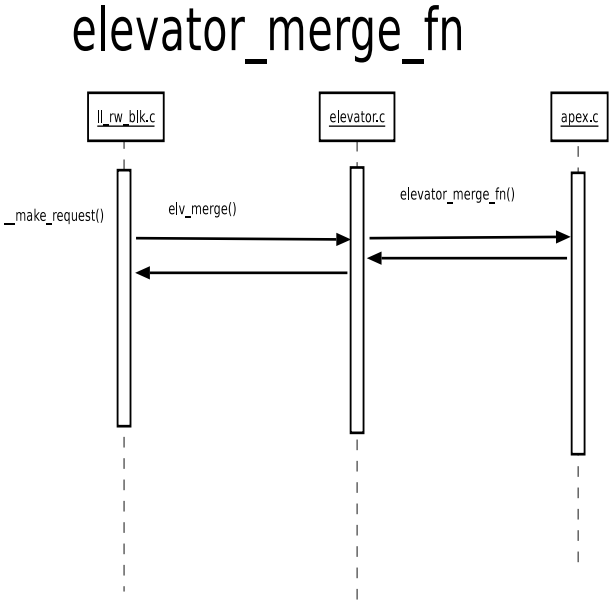
void elevator_merged_fn (request_queue_t *, struct request *)

Denne kalles etter at en forespørsel har blitt slått sammen med en annen forespørsel, for å oppdatere datastrukturen som holder rede på forespørslene.

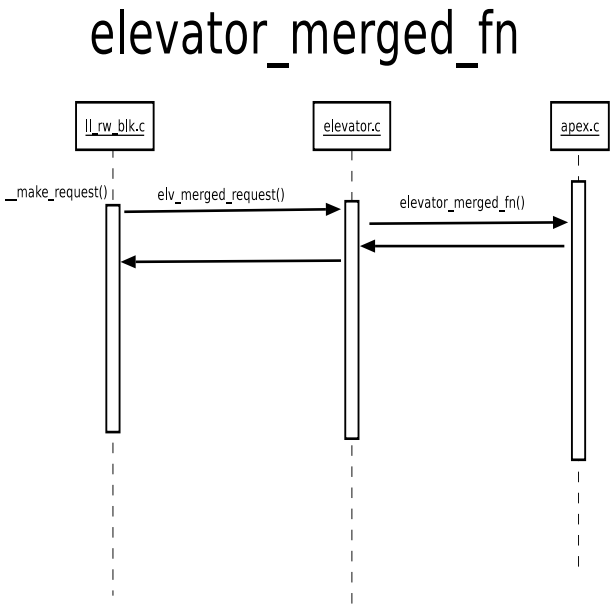
Denne funksjonen er implementert i skeduleren cfq, deadline io, og anticipatoryIO.



Figur A.1: Grensesnitt for disk-skedulering



Figur A.2: elevator_merge_fn()



Figur A.3: elevator_merged_fn()

void elevator_merge_req_fn (request_queue_t *, struct request *, struct request *)

Denne kalles for å slå sammen en forespørsel med en annen.
elevator_merge_req er implementert i alle disk-skedulererne.

struct request * elevator_next_req_fn (request_queue_t *)

Denne kalles for å hent neste forespørsel fra køen.
Denne funksjonen er implementert i alle disk-skedulererne.

void elevator_add_req_fn (request_queue_t *, struct request *, int)

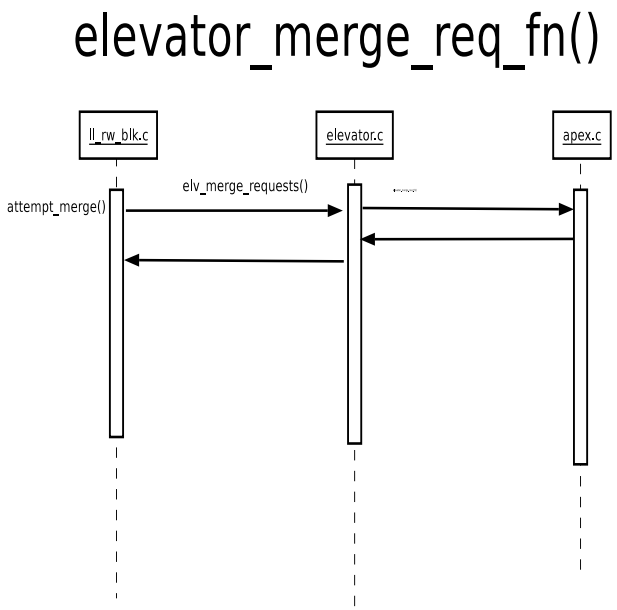
Denne kalles for å sette inn en forespørsel i køen. Det siste argumentet er et symbolsk tall som angir hvor i køen forespørselen settes inn.
Denne funksjonen er implementert i alle disk-skedulererne.

void elevator_remove_req_fn (request_queue_t *, struct request *);

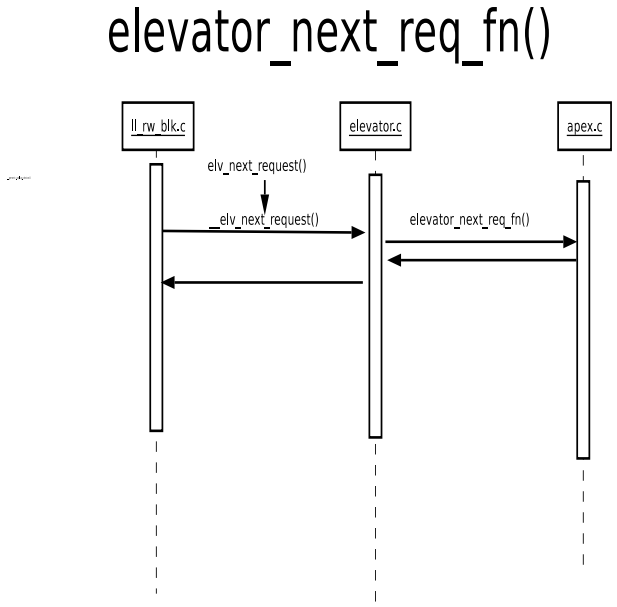
Denne kalles når driveren er ferdig med en forespørsel. Funksjonen er implementert i skedulererne cfq, deadline IO anticipatory IO.

void elevator_requeue_req_fn (request_queue_t *, struct request *)

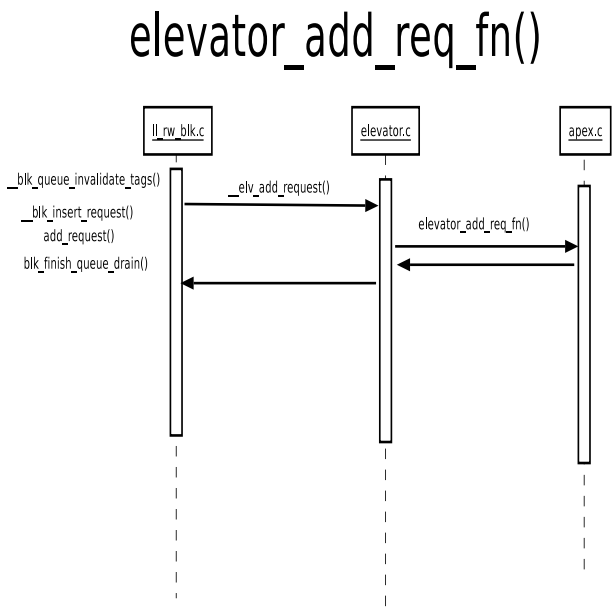
Denne kalles når en ubehandlet forespørsel skal legges inn i køen igjen, selv om den ikke er ny. Funksjonen er implementert i skedulererne CFQ og Anticipatory IO.



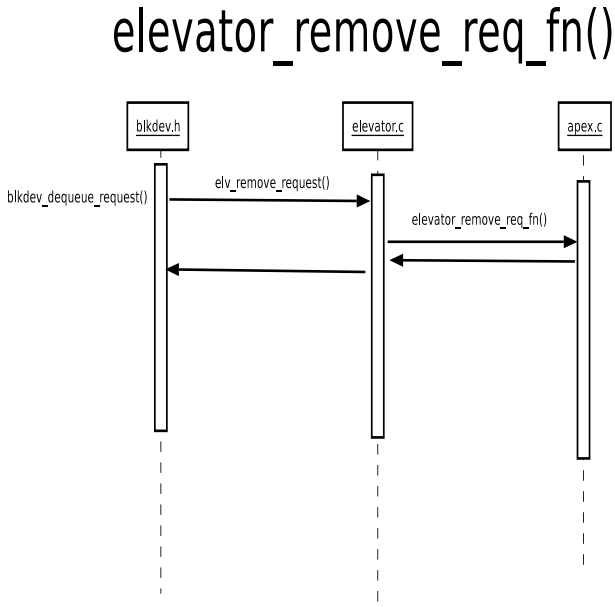
Figur A.4: elevator_merge_req_fn()



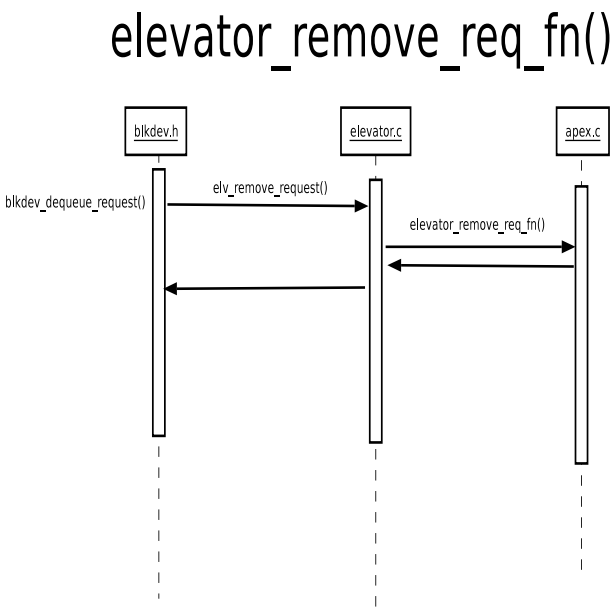
Figur A.5: elevator_next_req_fn()



Figur A.6: elevator_add_req_fn()



Figur A.7: elevator_remove_req_fn()



Figur A.8: elevator_requeue_req_fn()

int elevator_queue_empty_fn (request_queue_t *)

Denne funksjonen kalles for å undersøke om en kø er tom. Den er implementert i CFQ, deadline-io, og anticipatory IO skedulererne.

void elevator_completed_req_fn (request_queue_t *, struct request *)

Denne kalles når en forespørsel er blitt utført. Den er implementert i skedulererne CFQ og anticipatory IO.

struct request * elevator_former_req_fn (request_queue_t *, struct request *)

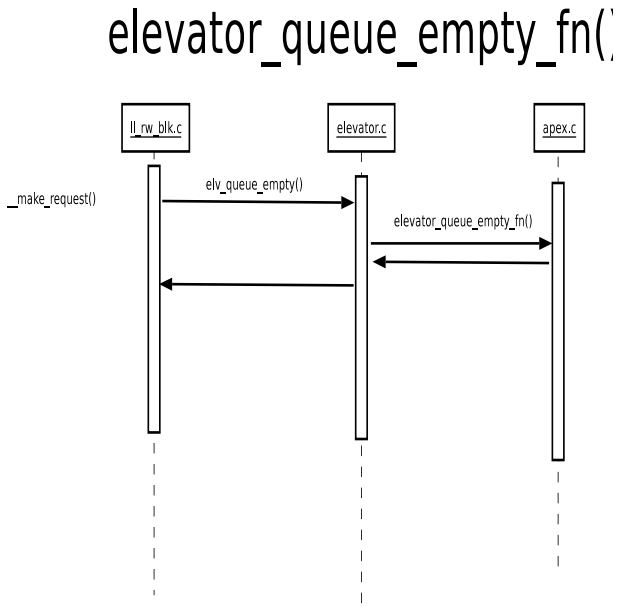
Denne kalles for å finne den forespørselen som kom først i køen i forhold til plasseringen av forespørselen som sendes inn. Funksjonen er implementert i skedulererne deadline IO, anticipatory IO og CFQ.

struct request * elevator_latter_req_fn (request_queue_t *, struct request *)

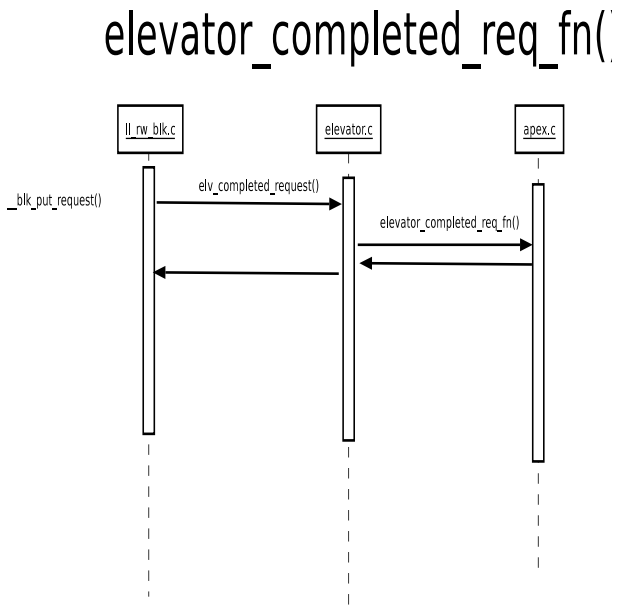
Denne kalles for å finne den forespørselen som kom sist i køen i forhold til plasseringen av forespørselen som sendes inn. Funksjonen er implementert i skedulererne deadline IO, anticipatory IO og CFQ.

int elevator_set_req_fn (request_queue_t *, struct request *, int)

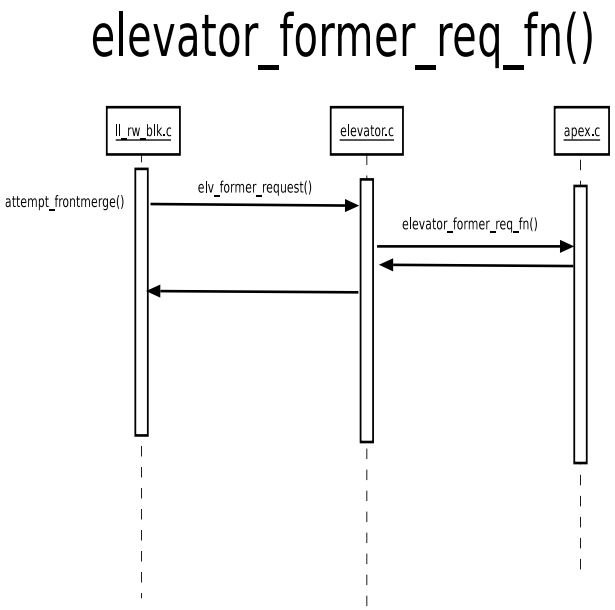
Denne kalles for å allokere et forespørselsobjekt på en kø. Det siste argumentet er et symbolsk tall som angir bitmap for enheten. Funksjonen er implementert i skedulererne CFQ, deadline io, og anticipatory IO.



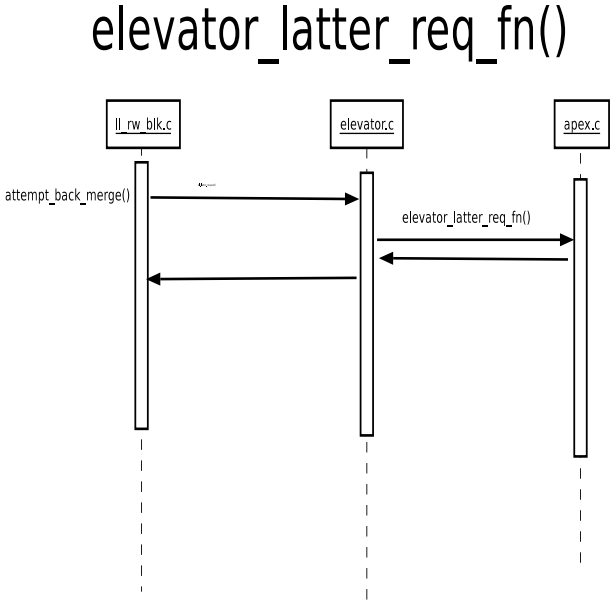
Figur A.9: elevator_queue_empty_fn()



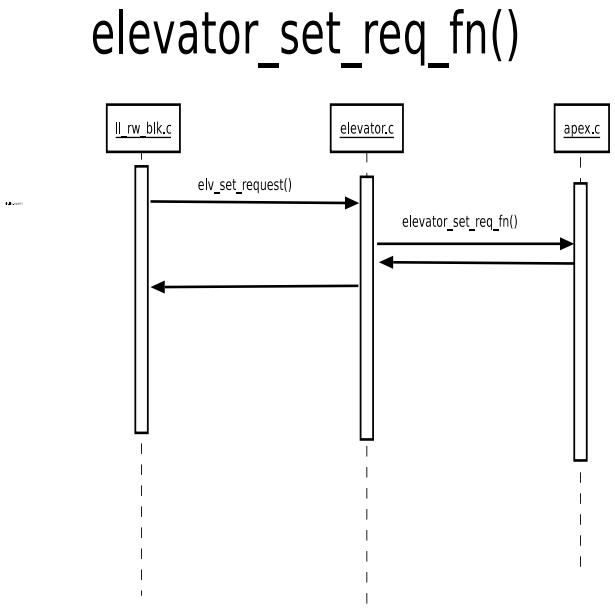
Figur A.10: elevator_completed_req_fn()



Figur A.11: elevator_former_req_fn()



Figur A.12: elevator_latter_req_fn()



Figur A.13: elevator_set_req_fn()

void elevator_put_req_fn (request_queue_t *, struct request *)

Denne kalles for å legge en forespørsel på en kø. Den er implementert i skedulererne deadline IO, anticipatory IO og CFQ.

int elevator_may_queue_fn (request_queue_t *, int)

Denne kalles for å finne ut om en kø kan ta imot en forespørsel. Den er implementert i skedulererne anticipatory IO og CFQ.

int elevator_init_fn (request_queue_t *, elevator_t *)

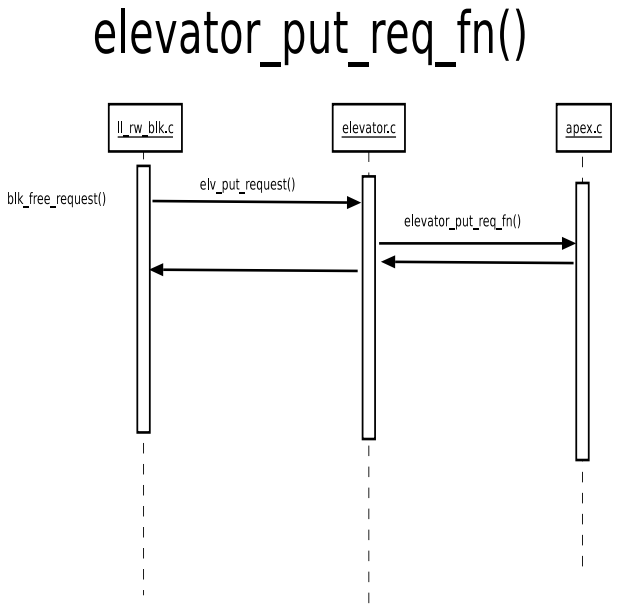
Denne kalles for å sette opp datastrukturen til en disk-skedulerer. Den er implementert i deadline IO, anticipatory IO og CFQ.

void elevator_exit_fn (elevator_t *)

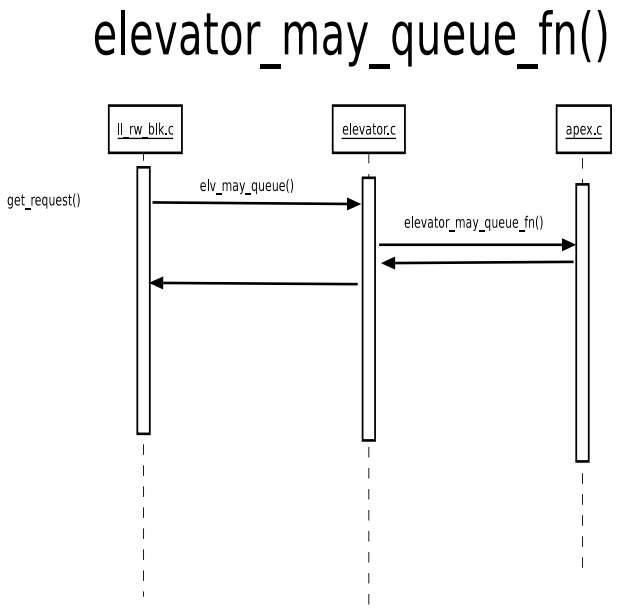
Denne kalles for å rive ned datastrukturen til en disk-skedulerer, for eksempel når man bytter fra en skeduleringsalgoritme til en annen. Den er implementert i deadline IO, anticipatory IO, og CFQ.

A.1.2 Implementasjon

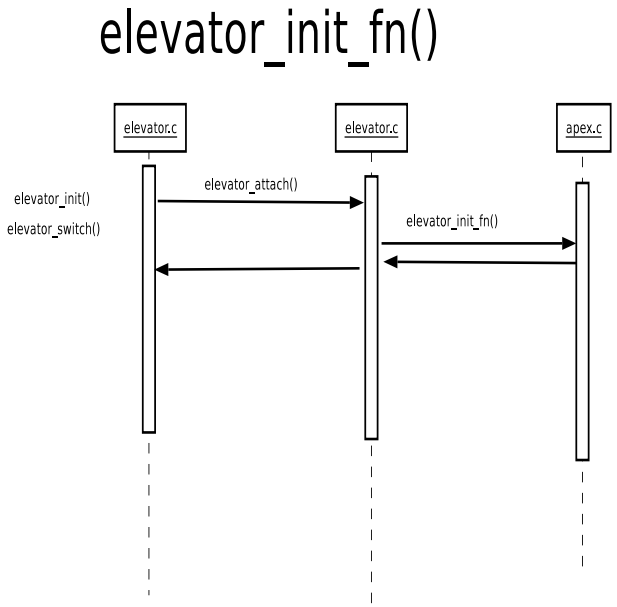
Koden som implementerer rammeverket ligger hovedsaklig i */drivers/block/ll_rw_blk.c* og i */drivers/block/elevator.c*. I *elevator.c* finnes også mange biblioteksfunksjoner for disk skedulering, blant annet standardimplementasjoner av grensesnittfunksjonene vi så på over. Disse standardfunksjonene har prefix *elv*, for eksempel *elv_former_request*.



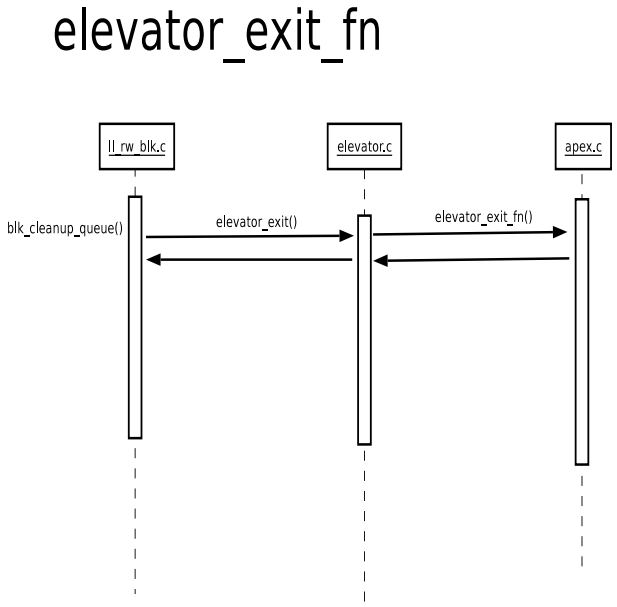
Figur A.14: elevator_put_req_fn()



Figur A.15: elevator_may_queue_fn()



Figur A.16: elevator_init_fn()



Figur A.17: apex_exit_fn()

A.1.3 Implementasjon i forskjellige diskskedulerere

Her er en oversikt som viser hvilke av de seksten grensesnittoperasjonene som er implementert i de forskjelilge skedulererne:

Linus elevator (noop) - fire operasjoner

- elevator_merge_fn
- elevator_merge_req_fn
- elevator_next_req_fn
- elevator_add_req_fn

Deadline IO - tretten operasjoner

- elevator_merge_fn
- elevator_merged_fn
- elevator_merge_req_fn
- elevator_next_req_fn
- elevator_add_req_fn
- elevator_remove_req_fn
- elevator_queue_empty_fn
- elevator_former_req_fn
- elevator_latter_req_fn
- elevator_set_req_fn
- elevator_put_req_fn
- elevator_init_fn
- elevator_exit_fn

Anticipatory IO - seksten operasjoner

- elevator_merge_fn
- elevator_merged_fn
- elevator_merge_req_fn
- elevator_next_req_fn
- elevator_add_req_fn
- elevator_remove_req_fn
- elevator_requeue_req_fn
- elevator_queue_empty_fn
- elevator_completed_req_fn

- elevator_former_req_fn
- elevator_latter_req_fn
- elevator_set_req_fn
- elevator_put_req_fn
- elevator_may_queue_fn
- elevator_init_fn
- elevator_exit_fn

CFQ - seksten operasjoner

- elevator_merge_fn
- elevator_merged_fn
- elevator_merge_req_fn
- elevator_next_req_fn
- elevator_add_req_fn
- elevator_remove_req_fn
- elevator_requeue_req_fn
- elevator_queue_empty_fn
- elevator_completed_req_fn
- elevator_former_req_fn
- elevator_latter_req_fn
- elevator_set_req_fn
- elevator_put_req_fn
- elevator_may_queue_fn
- elevator_init_fn
- elevator_exit_fn

A.2 Eksport av parametre til sysfs

Parametere og statusinformasjon om disk-skedulererne kan eksponeres med skrive- eller leserettigheter til sysfs. Konvensjonen er at disse eksponeres under
`/sys/block/<disk>/queue/iosched/` . Man kan skrive og lese til disse filene fra kommandolinjen, for eksempel ved hjelp av `cat` og `echo` .

Lese- og skriveoperasjonene til en disk-skedulerer er definert i feltet `.elevator_ktype\`i structen `elevator_type` , som inneholder feltene `.sysfs_ops` og `default_attrs` . Lagring og visning gjøres av funksjonene som er lagret i de enkelte attributtenes felter `show` og `store` .

For å gjøre et attributt lesbart og skrivbart gjennom sysfs, definerer man en struct med felter for filentry, en show-funksjon og en store-funksjon. Denne structen legges så inn i listen med attributter `default_attribs` i elevatorens `elevator_ktype` .

A.3 Datamodell

Alle data som er assosiert med en elevator lagres i feltet `elevator_data` i elevator-objektet. Dette feltet fylles opp når `elevator_init_fn` kalles. Her ligger alle konfigurasjonsparametre og køene med forespørsler.

Forespørsler lagres som en egen datatype, for eksempel `cfq_rq` i cfq, `as_rq` i anticipatory io. En forespørsel har typisk et `io_context` assosiert med seg, som inneholder informasjon om prosessen som har kommer med forespørselen. `io_context` kan ha flere varianter, feks `as_io_context` og `cfq_io_context` , som er definert i `blkdev.h` . Dersom man ønsker å lage sitt eget io-kontekst objekt må man skrive kode som ligger derfor utenfor kjernemodulen.

A.4 Lasting og bytte av disk-skedulerer som en kjernemodul

Gitt at man har skrevet og kompilert en kjernemodul `apex.ko` kan man laste den inn og få systemet til å bruke den på følgende måte:

```
debian\# insmod apex.ko
debian\# lsmod
Module                  Size  Used by
apex                    2816  0
ide\_scsi               15364  0
```

Informasjon om disk-skedulerere er eksportert til /sys/block/<enhet>:

```
debian\# cat /sys/block/hdb/queue/scheduler
noop [anticipatory] apex
```

Her er anticipatory skedulerer valgt. Slik bytter man:

```
debian\# echo apex > /sys/block/hdb/queue/scheduler

debian\# cat /sys/block/hdb/queue/scheduler
noop anticipatory [apex]}
```

Systemet kjører nå med apex-skedulereren på /dev/hdb.

A.5 Kommentarer til kildekoden

I denne seksjonen kommenterer vi utvalgte deler av kildekoden for å illustrere virkemåten i detalj.

Koden til APEX er å finne i filen *apex.c*. Det er en kjernemodul som kan lastes under kjøretid. For å tilpasse modulen til resten av kildekoden i Linux er det også gjort en del tilføyelser i */include/linux/block/blk-dev.h*, stort sett innføringen av datatypen *apex_io_context*, et objekt som holder ekstra informasjon om en forespørsel. Koden er basert på skedulereren CFQ. Store deler av koden som finnes i *apex.c* er derfor lik den man finner i *drivers/block/cfq-iosched.c*.

A.5.1 Logge-fasilitet

For å debugge og logge kjøringen av skedulereren bruker vi funksjonen *printk*, som logger meldinger til */var/log/messages* og */var/log/kern.log*. For å skru av og på slik logging er det definert to symboler:

```
/*
 * EXTRA:  Debugging and logging.
 */
#define APEX_DEBUG
#define APEX_TRACE
```

Symbolet *APEX_DEBUG* styrer logging som brukes til debugging, og *APEX_TRACE* brukes for å styre logging som brukes for testformål. Logging underveis i kildekoden omsluttes av en test på om disse symbolene er definert, slik:

```
       #ifdef APEX_DEBUG
         printk("End of round: %lu now: %lu ", apexd->end_of_round, now);
         printk("Requests in driver: %d\n", apexd->rq_in_driver);
       #endif /* DEBUG */
```

A.5.2 Kollibyggeren

Kollibyggeren er implementert i *apex_dispatch_requests*, som kalles av *apex_next_request*, som kalles når driveren ber om en ny forespørsel, og i *apex_insert_request*, som kalles når en ny forespørsel ankommer skedulereren.

Det første som skjer er at vi tester om det er mulig å bygge et kolli:

```
       /*
        * Return 0 when end_of_round is ok and in the future.
        *
        */
       if (list_empty(&apexd->rr_list) || (apexd->rq_in_driver > 1) ||
           (time_before(now, apexd->end_of_round)
            &&
            time_after(now + (apexd->tes * apexd->rq_in_driver), apexd->end_of_round)))
         return 0;

       #ifdef APEX_DEBUG
          printk("Doing a round.\n");
       #endif /* DEBUG */
```

Som vi ser så må det være forespørsler som venter på tur i *rr_list*, og det må ikke være mer enn en forespørsel i driveren. Inneværende runde må enten være over, eller så må det være tid nok til flere forespørsler i inneværende runden.

Dersom vi kan bygge et nytt kolli, oppdaterer vi tokenverdiene for køene som har forespørsler, slik:

```
        /* NOTE: Update all tokens before select */
        list_for_each_safe(entry, tmp, &apexd->rr_list) {
          apexq = list_entry_apexq(entry);
          apex_update_tokens(apexq);
        }
```

Her kalles *apex_update_tokens*, og den ser slik ut:

```
/* NOTE: Update tokens & token_update each time queue is on rr_list */

static void apex_update_tokens(struct apex_queue *apexq)
{
       unsigned long now, since_last_update;

       now = jiffies;

       /* NOTE: If token_update is in the future. */
       if(now < apexq->token_update)
         {
           apexq->tokens = apexq->b;
           apexq->token_update = now;
           printk("HZ overflow");
         }
       else
         {
           since_last_update = now - apexq->token_update;
           if (since_last_update > (HZ / apexq->r))
             {
               apexq->tokens = apexq->tokens + ((since_last_update*apexq->r)/HZ);
               if (apexq->tokens > apexq->b)
                 apexq->tokens = apexq->b;
               apexq->token_update = now;
             }
         }
}
```

Denne funksjonen inneholder først en test på det usannsynlige tilfellet at *HZ*, som angir tiden i millisekunder, går over sin maksimalverdi og blir 0 igjen. Den er av typen unsigned long.

Deretter sjekker vi om det har gått lang nok tid siden forrige oppdatering, og hvis det er det får køen så mange tokens som køens *r*-verdi tilsier. Til slutt sjekker vi at ikke antall tokens går over køens *b*-verdi.

Etter at vi har oppdatert alle tokens, finner vi ut når inneværende runde er slutt. Det gjøres slik:

```
        /* NOTE: End of current round is the default deadline.
         *       If round is over, start at 20x batch size.
         */
        if (time_before(now, apexd->end_of_round))
          earliest_deadline = apexd->end_of_round - now;
        else
          earliest_deadline = 20*apexd->tes;

        tstart = now;
        list_for_each_safe(entry, tmp, &apexd->rr_list)
          {
            apexq = list_entry_apexq(entry);

            if(apexq->queue_type == APEX_QT_RT)
              {
                this_deadline = apex_next_deadline(apexq);
                if(this_deadline &&
                   (time_before(this_deadline, earliest_deadline + tstart)))
                  {
                    if (this_deadline > tstart)
                      earliest_deadline = this_deadline - tstart;
                    else
                      earliest_deadline = apexd->tes;
                  }
              }
          }

        printk("EXTRA DEADLINE AFTER: %lu\n", earliest_deadline);

        batch_size = (earliest_deadline/apexd->tes);

        /* Minimum of 1 in the batch */
```

```
        if (batch_size == 0)
          batch_size = 1;

        #ifdef APEX_TRACE
          printk("Roundtime: %lu\n", earliest_deadline);
          printk("Roundsize: %d\n", batch_size);
        #endif

        apexd->end_of_round = tstart + earliest_deadline;

        #ifdef APEX_DEBUG
          printk("Earliest deadline: In %lu ms, batch size: %d.\n", earliest_deadline, batch_size);
        #endif /* DEBUG */
```

Dersom vi er midt i en runde, tar vi slutten på inneværende runde som utgangspunkt, ellers bruker vi behandlingstiden for 20 forespørsler. Når vi har et utgangspunkt, går vi igjennom alle sanntidskøene og ser om det er noen av dem som har tidsfrist før denne rundens slutt, og justerer rundeslutten tilsvarende. Dersom en tidsfrist er brutt, sier vi at runden skal slutte på den tiden det tar å gjennomføre en forespørsel. Et kolli består alltid av minst en forespørsel.

Når vi vet hvor tidspunktet for når inneværende runde slutter, går vi først igjennom sanntidskøene og velger ut de forespørslene som må sendes avgårde for ikke å bryte tidsfrister. I denne første omgangen tar vi ikke hensyn til køenes tokenverdier.

```
        queued = 0;

        /* First round: Realtime queues
         * We first select from rt-queues so as not to break any deadlines.
         * Then we select from the queues based on tokens.
         */

        list_for_each_safe(entry, tmp, &apexd->rr_list)
          {
                apexq = list_entry_apexq(entry);

                if(apexq->queue_type == APEX_QT_RT)
                  {
                    while ((queued <= batch_size)
                           && (!RB_EMPTY(&apexq->sort_list))
                           && time_before_eq(apex_next_deadline(apexq), apexd->end_of_round))
                      {
                        apex_dispatch_request(q, apexd, apexq);
                        queued++;
                        if (apexq->tokens)
                          apexq->tokens--;
                    #ifdef APEX_DEBUG
                      printk("Deadline round, no tokens: %d queued. pid %lu %d tokens.\n",
                             queued, apexq->key, apexq->tokens);
                    #endif /* DEBUG */
                      }
                  }
          }
```

Her kalles *apex_next_deadline*, som tar en kø som innparameter og returnerer tidsfristen til forespørselen som ligger først i FIFO-køen:

```
static unsigned long apex_next_deadline(struct apex_queue * apexq)
{
  unsigned long this_deadline = 0;
  struct apex_rq *crq;
  /* Next read-request: */
  if (!list_empty(&apexq->fifo[0]))
    {
      crq = RQ_DATA(list_entry(apexq->fifo[0].next, struct request, queuelist));
      this_deadline = crq->queue_start + apexq->deadline;
    }
  else
    /* Next write-request */
    if (!list_empty(&apexq->fifo[1]))
      {
        crq = RQ_DATA(list_entry(apexq->fifo[1].next, struct request, queuelist));
        this_deadline = crq->queue_start + apexq->deadline;
      }
```

```
  return this_deadline;
}
```

Køene har en FIFO-kø for leseforespørsler, og en for skriveforespørsler. Vi velger tidspunktet for tidsfristen fra den køen som har ventende forespørsler. Tidsfristen er tidspunktet for når forespørselen kom inn i køen pluss køens tidsfrist.

Tilbake i *apex_dispatch_requests* så har vi altså nettopp valgt ut de forespørslene som må tas for ikke å bryte tidsfrister. Deretter går vi igjennom sanntidskøene på nytt, og velger ut forespørsler basert på hvor mange tokens køen har:

```
        list_for_each_safe(entry, tmp, &apexd->rr_list)
          {
                apexq = list_entry_apexq(entry);

                if(apexq->queue_type == APEX_QT_RT)
                  {
                    while((apexq->tokens > 0) &&  queued<= batch_size && (!RB_EMPTY(&apexq->sort_list)))
                      {
                        apex_dispatch_request(q, apexd, apexq);
                        apexq->tokens--;
                        queued++;
                      }
                    #ifdef APEX_DEBUG
                      printk("Deadline round, with tokens: %d queued. pid %lu %d tokens.\n",
                             queued, apexq->key, apexq->tokens);
                    #endif /* DEBUG */
                  }
          }
```

Når vi er ferdig med sanntidskøene gjøres det samme med prioritetskøene:

```
        /* Second round: Rest of priority queues. */
        list_for_each_safe(entry, tmp, &apexd->rr_list)
          {
                apexq = list_entry_apexq(entry);

                if(apexq->queue_type == APEX_QT_PRI)
                  {
                    while((apexq->tokens > 0) && queued<= batch_size && (!RB_EMPTY(&apexq->sort_list)))
                      {
                        apex_dispatch_request(q, apexd, apexq);
                        apexq->tokens--;
                        queued++;
                      }

                #ifdef APEX_DEBUG
                  printk("Token round: %d queued. pid %lu %d tokens.\n",
                         queued, apexq->key, apexq->tokens);
                #endif /* DEBUG */
                  }
          }
```

Dersom det fortsatt er plass igjen i inneværende kolli, velger vi forespørsler fra best-effort køene, basert på hvor mange tokens disse køene har:

```
        /* Third round: Best effort queues. */
        list_for_each_safe(entry, tmp, &apexd->rr_list)
          {
                apexq = list_entry_apexq(entry);

                if(apexq->queue_type == APEX_QT_BE)
                  {
                    while((apexq->tokens > 0) && queued<= batch_size && (!RB_EMPTY(&apexq->sort_list)))
                      {
                        apex_dispatch_request(q, apexd, apexq);
                        queued++;
                      }
                  #ifdef APEX_DEBUG
                    printk("Best effort round: %d queued. pid %lu %d tokens.\n",
                           queued, apexq->key, apexq->tokens);
                  #endif /* DEBUG */
```

```
                }
        }
```

Til slutt tar vi hånd om et spesialtilfelle: Disken har kapasitet og vi har forespørsler i kø, men ingen forespørsler er blitt valgt ut fra køene, fordi ingen køer har tokens. I dette tilfellet velger vi en forespørsel fra hver kø helt til vi har valgt ut så mange forespørsler som det er plass til i inneværende runde.

```
       /* Last round: None of the queues have tokens, and no dispatch yet.
        *             In this case we dispatch one request from each batch,
        *             limited by the size of the current batch.
        */
       if (queued == 0)
         {
           list_for_each_safe(entry, tmp, &apexd->rr_list)
             {
               apexq = list_entry_apexq(entry);
               if(queued<= batch_size && (!RB_EMPTY(&apexq->sort_list)))
                 {
                   apex_dispatch_request(q, apexd, apexq);
                   printk("Minimum of one from each queue dispatched.");
                   queued++;
                 }
             }
         }

         #ifdef APEX_TRACE
            printk("Queued: %d\n", queued);
         #endif


       return queued;
}
```

apex_dispatch_requests returnerer antall forespørsler som ble valgt ut.

A.5.3 Eksponering av informasjon i sys-fs

Vi bruker sys-fs for å gjøre diverse informasjon om APEX løpende tilgjengelig for brukerne. Dette er et eksempel på en slik funksjon, den viser frem en liste over alle køene som kjører akkurat nå, med tilhørende informasjon.

```
static ssize_t apex_queueinfo_show(struct apex_data *apexd, char *page)
{
  ssize_t len = 0;

  struct list_head *entry, *tmp;
  list_for_each_safe(entry, tmp, &apexd->rr_list) {
    struct apex_queue *apexq = list_entry_apexq(entry);
    len += sprintf(page+len, "busy:%lu ", apexq->key);
    if (apexq->queue_type == APEX_QT_RT)
      len += sprintf(page, "(RT) ");
    if (apexq->queue_type == APEX_QT_PRI)
      len += sprintf(page, "(PR) ");
    if (apexq->queue_type == APEX_QT_BE)
      len += sprintf(page, "(BE) ");

    len += sprintf(page+len, "r:%u b:%d tokens:%d  deadline:%lu token_update:%lu\n",
                   apexq->r, apexq->b, apexq->tokens,
                   apexq->deadline, jiffies - apexq->token_update);
  }

  list_for_each_safe(entry, tmp, &apexd->empty_list) {
    struct apex_queue *apexq = list_entry_apexq(entry);
    len += sprintf(page+len, "empty:%lu ", apexq->key);
    if (apexq->queue_type == APEX_QT_RT)
      len += sprintf(page+len, "(RT) ");
    if (apexq->queue_type == APEX_QT_PRI)
      len += sprintf(page+len, "(PR) ");
    if (apexq->queue_type == APEX_QT_BE)
      len += sprintf(page+len, "(BE) ");

    len += sprintf(page+len, "r:%u b:%d tokens:%d deadline:%lu token_update:%lu\n",
                   apexq->r, apexq->b, apexq->tokens,
```

```
                    apexq->deadline, jiffies - apexq->token_update);
  }
  len += sprintf(page+len, "\n");

  return len;
}
```

A.6 Script brukt under testing

Her er scriptene som ble brukt under testing av APEX og de andre disk-skedulererne.

A.6.1 APEX

Hovedscriptet er *test.sh*, som starter en egen prosess for de andre scriptene:

```
#!/bin/bash

./big1.py &
./big2.py &

./real-01.py &
./real-02.py &
./real-03.py &
./real-04.py &
./real-05.py &

./real-06.py &
./real-07.py &
./real-08.py &
./real-09.py &
./real-10.py &

./pri-01.py &
./pri-02.py &
./pri-03.py &
./pri-04.py &
./pri-05.py &

./pri-06.py &
./pri-07.py &
./pri-08.py &
./pri-09.py &
./pri-10.py &
```

Vi har eksponert grensesnittet for å sette parametre for tjenestekvalitet i filen *scheduler.py*:

```
def set_r(value):
  r = open('/sys/block/hda/queue/iosched/apex_r','w')
  r.write(value)

def set_b(value):
  r = open('/sys/block/hda/queue/iosched/apex_b','w')
  r.write(value)

def set_deadline(value):
  r = open('/sys/block/hda/queue/iosched/apex_deadline','w')
  r.write(value)

def set_queuetype(value):
  r = open('/sys/block/hda/queue/iosched/apex_queuetype','w')
  r.write(value)
```

Det er to best-effort prosesser som leser kontinuerlig fra hver sine store filer og teller antall leste 64K-blokker. Her er scriptet som leser disse filene:

```
#!/usr/bin/python

import os
import time
import scheduler

print 'A pid', os.getpid()

start_time = time.time()
rate = 288
reads = 0

scheduler.set_r('100')
scheduler.set_b('1000')
scheduler.set_queuetype('BE')

logfilename = 'results/big-%d' %os.getpid()
logfile = open(logfilename, 'w')

f = open('/media/media/test/big01.mpg')
a = f.read(64000)
sofar = time.time() - start_time
while (sofar <= 600):
     sofar = time.time() - start_time
     a = f.read(64000)
     reads = reads + 1
     if (a == ''):
        print 'reread.'
        f = open('/media/media/test/big01.mpg')
f.close()

logfile.write('Reads: %d' %reads)
```

Det er ti sanntidsprosesser som leser ti-minutters videoer med en rate på 288KB/sekundet. Her er et slik script:

```
#!/usr/bin/python

import os
import time
import scheduler

print 'A pid', os.getpid()

start_time = time.time()
rate = 288
reads = 0
sleeps = 0

scheduler.set_r('5')
scheduler.set_b('100')
scheduler.set_deadline('50')
scheduler.set_queuetype('RT')

logfilename = 'results/real-%d' %os.getpid()
logfile = open(logfilename, 'w')

f = open('/media/media/test/01.mpg')
a = f.read(64000)
while (a!= ''):
  sofar = time.time() - start_time
  if ((reads*64/sofar) > rate):
     time.sleep(0.5)
     sleeps = sleeps + 1
  else:
     a = f.read(64000)
     reads = reads + 1
f.close()

logfile.write('Sleeps: %d' %sleeps)
```

Til slutt har vi ti prioritetsprosesser som leser hver sine ti-minutters videoer med en rate på 288KB. Scriptet som gjør dette ser slik ut:

```
#!/usr/bin/python

import os
import time
import scheduler

print 'A pid', os.getpid()

start_time = time.time()
rate = 288
reads = 0
sleeps = 0

scheduler.set_r('5')
scheduler.set_b('100')
scheduler.set_queuetype('PR')

logfilename = 'results/pri-%d' %os.getpid()
logfile = open(logfilename, 'w')

f = open('/media/media/test/p01.mpg')
a = f.read(64000)
while (a!= ''):
  sofar = time.time() - start_time
  if ((reads*64/sofar) > rate):
     time.sleep(0.5)
     sleeps = sleeps + 1
  else:
     a = f.read(64000)
     reads = reads + 1
f.close()

logfile.write('Sleeps: %d' %sleeps)
```

A.6.2 CFQ og øvrige skedulere

For CFQ og de øvrige skedulererne har vi to prosesser som leser kontinuerlig fra to store filer, og tyve prosesser som leser hver sine 10-minutters videoer med en rate på 288KB/sekundet.

Her er scriptet som starter opp disse prosessene:

```
#!/bin/bash

./big1.py &
./big2.py &

./real-01.py &
./real-02.py &
./real-03.py &
./real-04.py &
./real-05.py &

./real-06.py &
./real-07.py &
./real-08.py &
./real-09.py &
./real-10.py &

./pri-01.py &
./pri-02.py &
./pri-03.py &
./pri-04.py &
./pri-05.py &

./pri-06.py &
./pri-07.py &
./pri-08.py &
./pri-09.py &
./pri-10.py &
```

Her er scriptet for de to leseprosessene som leser fra store filer:

```
#!/usr/bin/python

import os
import time
import scheduler

print 'A pid', os.getpid()

start_time = time.time()
rate = 288
reads = 0

logfilename = 'results/big-%d' %os.getpid()
logfile = open(logfilename, 'w')

f = open('/media/media/test/big01.mpg')
a = f.read(64000)
sofar = time.time() - start_time
while (sofar <= 600):
     sofar = time.time() - start_time
     a = f.read(64000)
     reads = reads + 1
     if (a == ''):
        print 'reread.'
        f = open('/media/media/test/big01.mpg')
f.close()

logfile.write('Reads: %d' %reads)
```

Her er et script som leser småfilene:

```
#!/usr/bin/python

import os
import time
import scheduler

print 'A pid', os.getpid()

start_time = time.time()
rate = 288
reads = 0
sleeps = 0


logfilename = 'results/real-%d' %os.getpid()
logfile = open(logfilename, 'w')

f = open('/media/media/test/01.mpg')
a = f.read(64000)
while (a!= ''):
  sofar = time.time() - start_time
  if ((reads*64/sofar) > rate):
     time.sleep(0.5)
     sleeps = sleeps + 1
  else:
     a = f.read(64000)
     reads = reads + 1
f.close()

logfile.write('Sleeps: %d' %sleeps)
```

For å måle responstiden til alle skedulererne brukte vi det samme python-scriptet som over for de to best-effort prosessene, og et program skrevet i C for å kjøre de 20 sanntids- og prioritetsprosessene. C-programmet benytter RDTSC-instruksjonen for å måle tid. Denne instruksjonen finnes på Intel-prosessorer, og returnerer en *long long int* som teller klokkesykler. For å regne ut tidsbruk dividerer man antall klokkesykler som har gått med antall klokkesykler maskinen bruker per sekund.

Her er C-programmet som ble benyttet til disse testene:

```
#include <stdio.h>
#include <sys/time.h>
#include <time.h>


/*******************************
 *      QoS-interface          *
 *                             *
 *******************************/

void set_r(char* value)
{
  FILE *r_file = fopen("/sys/block/hda/queue/iosched/apex_r", "w");
  fprintf(r_file, "%s", value);
  close(r_file);
}

void set_b(char* value)
{
  FILE *b_file = fopen("/sys/block/hda/queue/iosched/apex_b", "w");
  fprintf(b_file, "%s", value);
  close(b_file);
}

void set_deadline(char* value)
{
  FILE *deadline_file = fopen("/sys/block/hda/queue/iosched/apex_deadline", "w");
  fprintf(deadline_file, "%s", value);
  close(deadline_file);
}

void set_queuetype(char* value)
{
  FILE *queuetype_file = fopen("/sys/block/hda/queue/iosched/apex_queuetype", "w");
  fprintf(queuetype_file, "%s", value);
  close(queuetype_file);
}

/*******************************
 *                             *
 *       RDTSC-interface       *
 *                             *
 *******************************/

__inline__ unsigned long long int get_rdtsc()
{
  unsigned long long int x;
  __asm__ volatile (".byte 0x0f, 0x31" : "=A" (x));
  return x;
}

long long int cycles_to_ms(long long int rdtsc)
{
  return (rdtsc * 1000/1329687444);
}

/*******************************
 *        Reading files        *
 *                             *
 *******************************/

void read_r_file(char *filename)
{
  printf("R-file.\n");

  set_r("5");
  set_b("100");
  set_deadline("200");
  set_queuetype("RT");

  char buf[64000];
```

```
  FILE *fp = fopen(filename, "r");
  char logfilename[100];
  FILE *logfile;
  int chars_read = 1;
  int buffers_read = 0;

  long long int read_start;
  long long int read_stop;

  long duration = 0;

  sprintf(logfilename, "/home/thenrikh/time-test/results-apex/R-%d", getpid());
  logfile = fopen(logfilename, "w");

  struct timeval start_time;
  struct timeval now;
  gettimeofday(&start_time, NULL);
  gettimeofday(&now, NULL);

  while (chars_read > 0)
    {
      gettimeofday(&now, NULL);
      int sofar = now.tv_sec - start_time.tv_sec;
      if(sofar && (buffers_read*64/sofar) > 288)
        {
          usleep(500000);
        }
      read_start = get_rdtsc();
      chars_read = fread(buf, 1, 64000, fp);
      read_stop = get_rdtsc();

      buffers_read++;
      duration = read_stop - read_start;
      fprintf(logfile, "%d\n", cycles_to_ms(duration));
    }

  printf("Buffers read by pid R-%d: %d\n", getpid(), buffers_read);
}

void read_p_file(char *filename)
{
  printf("P file.\n");

  set_r("5");
  set_b("100");
  set_queuetype("PR");

  char buf[64000];
  FILE *fp = fopen(filename, "r");
  char logfilename[100];
  FILE *logfile;
  int chars_read = 1;
  int buffers_read = 0;

  long long int read_start;
  long long int read_stop;

  long duration = 0;

  sprintf(logfilename, "/home/thenrikh/time-test/results-apex/P-%d", getpid());
  logfile = fopen(logfilename, "w");

  struct timeval start_time;
  struct timeval now;
  gettimeofday(&start_time, NULL);
  gettimeofday(&now, NULL);

  while (chars_read > 0)
    {
      gettimeofday(&now, NULL);
      int sofar = now.tv_sec - start_time.tv_sec;
      if(sofar && (buffers_read*64)/sofar > 288)
        {
          usleep(500000);
        }

      read_start = get_rdtsc();
      chars_read = fread(buf, 1, 64000, fp);
      read_stop = get_rdtsc();

      buffers_read++;
      duration = read_stop - read_start;
      fprintf(logfile, "%d\n", cycles_to_ms(duration));
    }
```

```
  printf("Buffers read by pid P-%d: %d\n", getpid(), buffers_read);
}

main()
{

  if(fork() == 0)
    {
      read_r_file("/media/media/test/01.mpg");
      exit(0);
    }

  if(fork() == 0)
    {
      read_r_file("/media/media/test/02.mpg");
      exit(0);
    }

  if(fork() == 0)
    {
      read_r_file("/media/media/test/03.mpg");
      exit(0);
    }

  if(fork() == 0)
    {
      read_r_file("/media/media/test/04.mpg");
      exit(0);
    }

  if(fork() == 0)
    {
      read_r_file("/media/media/test/05.mpg");
      exit(0);
    }

  if(fork() == 0)
    {
      read_r_file("/media/media/test/06.mpg");
      exit(0);
    }

  if(fork() == 0)
    {
      read_r_file("/media/media/test/07.mpg");
      exit(0);
    }

  if(fork() == 0)
    {
      read_r_file("/media/media/test/08.mpg");
      exit(0);
    }

  if(fork() == 0)
    {
      read_r_file("/media/media/test/09.mpg");
      exit(0);
    }

  if(fork() == 0)
    {
      read_r_file("/media/media/test/10.mpg");
      exit(0);
    }

  if(fork() == 0)
    {
      read_p_file("/media/media/test/p01.mpg");
      exit(0);
    }


  if(fork() == 0)
    {
      read_p_file("/media/media/test/p02.mpg");
      exit(0);
    }

  if(fork() == 0)
    {
      read_p_file("/media/media/test/p03.mpg");
      exit(0);
```

```
    }

  if(fork() == 0)
    {
      read_p_file("/media/media/test/p04.mpg");
      exit(0);
    }

  if(fork() == 0)
    {
      read_p_file("/media/media/test/p05.mpg");
      exit(0);
    }

  if(fork() == 0)
    {
      read_p_file("/media/media/test/p06.mpg");
      exit(0);
    }

  if(fork() == 0)
    {
      read_p_file("/media/media/test/p07.mpg");
      exit(0);
    }

  if(fork() == 0)
    {
      read_p_file("/media/media/test/p08.mpg");
      exit(0);
    }

  if(fork() == 0)
    {
      read_p_file("/media/media/test/p09.mpg");
      exit(0);
    }

  if(fork() == 0)
    {
      read_p_file("/media/media/test/p10.mpg");
      exit(0);
    }
}
```

Bibliografi

[1] Martin C. Narayanan P. S. Ozden B. Rastogi R. Silberschatz A. The fellini multimedia storage system. *Journal of Digital Libraries*, 1997.

[2] Govindan R. Anderson R., Osawa Y. A file system for continuous media. *Transactions on Computer Systems*, 1992.

[3] D. Parulkar. G. M. Buddhikot M. Chen X. J. Wu. Enhancements to 4.4bsd unix for efficient networked multimedia in project mars. *IEEE International Conference on Multimedia Computing and Systems (ICMCS'98)*, 1998.

[4] Livny M. Carey M. J, Jauhari R. Priority in dbms resource scheduling.

[5] Zakhor A. Chang E. Cost analysis for vbr video servers. *IEEE Multimedia, Vol4, No. 3, 1996*, 1996.

[6] Strosnider J. K. Daigle S. J. Disk scheduling for multimedia data streams. *Conference on Hich-Speed networking and Multimedia Computing, San Jose, CA, USA*, 1994.

[7] J Denning, P. Effects of scheduling on file memory operations. In *AFIPS pp.9-21*, April 1967.

[8] Daniel S. Geist R. A continuum of disk scheduling algoritms. *ACM Transactions on Computer Systems*, 1987.

[9] J Gemmel D. Multimedia network file servers: Multi-channel dalay sensitive data retrieval. *ACM Multimedia '93 Anaheim CA USA*, 1993.

[10] Sitaram Iyer and Peter Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *18th ACM Symposium on Operating Systems Principles*, October 2001.

[11] Layland J. W Liu C. Sceduling algorithms for multiprogramming in hard-real-time environment. *Journal of the ACM*, 1973.

[12] Robert Love. *Linux Kernel Developement*. Sams Publishing, 2003.

[13] Ketil Lund. *Adaptive Scheduling in a Multimedia DBMS*. PhD thesis, University of Oslo, 2003.

[14] G Merten, A. Some quantative techniques for file organization. Technical report, Wisconsin, USA, 1970.

[15] Wyllie J. Narashimha Reddy A. Disk scheduling in a multimedia i/o system.

[16] Pål Halvorsen Carsten Griwodz Ketil Lund Vera Goebel Thomas Plagemann. Storage systems support for multimedia application. Technical report, Department of Informatics, University of Oslo, 2003.

[17] Prashant J. Shenoy and Harrick M. Vin. Cello: A disk scheduling framework for next generation operating systems. In *Cello: A Disk Scheduling Framework for Next Generation Operating Systems*, 1998.

[18] Venkat Rangan P. Vin H.M. Designning a multi-user hdtv storage server. *IEEE Journal on Selected Areas in Communication*, 1996.

[19] Narasimha Reddy A.L Wijayaratne, r. Providing qos guarantees for disk i/o. *Springer Journal on Multimedia Systems*, 2000.

[20] Kandlur D. D Yu P. S., Chen M.S. Design and analysis of a grouped sweeping scheme for multimedia storage management. *Third International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV'92), La Jolla, Ca, USA,*, 1992.